Maisterintutkielma

Tietojenkäsittelytieteen maisteriohjelma

SPA-sovellukset hajautettuna järjestelmänä

Juha-Pekka Eloranta

3.5.2020

Matemaattis-luonnontieteellinen tiedekunta

Helsingin yliopisto

Ohjaaja(t)

Dr. M. Luukkainen, Prof. T. Mikkonen

Tarkastaja(t)

Prof. T. Mikkonen

Yhteystiedot

PL 68 (Pietari Kalmin katu 5)
00014 Helsingin yliopisto

Sähkopostiosoite: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen maisteriohjelma

Tekijä — Författare — Author

Juha-Pekka Eloranta

Työn nimi — Arbetets titel — Title

SPA-sovellukset hajautettuna järjestelmänä

Ohjaajat — Handledare — Supervisors

Dr. M. Luukkainen, Prof. T. Mikkonen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Maisterintutkielma	3.5.2020	65 sivua

Tiivistelmä — Referat — Abstract

Single-page application -mallista (SPA) on tullut suosittu tapa tehdä web-sovelluksia. Sen ansiosta web-sivujen käyttökokemuksesta saadaan enemmän työpöytäsovellusten kaltainen, kun jokaista sivuvaihtoa ja operaatiota varten ei tarvitse ladata palvelimelta uutta HTML-sivua.

SPA-malli tuo kuitenkin tavallisiin web-sovelluksiin mukaan hajautetun datan käsittelyn ongelmia. Eheyden hallinta hajautetuissa järjestelmissä on ikuisuusaihe tietojenkäsittelytieteen tutkimuksessa. SPA-sovelluksissa tähän ei kuitenkaan kiinnitetä juuri huomioita. Tässä tutkielmassa vertaillaan SPA-sovelluksia hajauttuihin tietokantoihin ja pyritään löytämään niistä tekniikoita, joita voisi soveltaa myös SPA-sovelluksiin.

Vertailu aloitetaan esittelemällä erilaisia hajautettuja tietokantoja ja arvioidaan mitkä niistä vastaavat eniten SPA-mallin tilannetta. Tämän jälkeen vertailu keskittyy kahteen osa-alueeseen. Ensin esitellään hajautettujen tietokantojen käyttämiä tapoja replikoida dataa pisteestä toiseen ja vertaillaan niihin web-sovelluksien tapoja siirtää dataa selaimen ja palvelimen välillä. Lisäksi tutkitaan kuinka eheyteen liittyvät mallit, kuten ACID-ominaisuudet sekä eristyvyystasot ja -anomaliat toteutuvat tai ilmenevät SPA-sovelluksissa.

Vertailun tuloksena havaittiin, että hajautettujen tietokantojen ja web-sovellusten käyttämät tiedonsiirtomenetelmät ovat jokseenkin erilaisia. Hajautetuissa tietokannoissa suositaan push-mallia päivitysten levittämisessä ja replikointi on sovelluskehittäjän näkökulmasta automaattista. Web-sovellukset käyttävät usein pull-mallia ja replikoinnin toteuttaminen on sovelluskehittäjän vastuulla. Eheyden hallintaan liittyen nousi esille joukko anomalioita, jotka ovat mahdollisia SPA-sovelluksissa ja vaativat huolellisuutta ohjelmoijalta. Löydösten perusteella on kuitenkin hyvät edellytykset lähteä kehittämään uusia kirjastoja, joilla havaittuja SPA-sovellusten ongelmia voitaisiin ratkaista.

ACM Computing Classification System (CCS)
Information systems → Parallel and distributed DBMSs
Information systems → Web applications

Avainsanat — Nyckelord — Keywords

SPA-sovellukset, hajautetut järjestelmät

Säilytyspaikka — Förvaringsställe — Where deposited

Helsingin yliopiston kirjasto

Muita tietoja — övriga uppgifter — Additional information

Ohjelmistojärjestelmien opintosuunta

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Juha-Pekka Eloranta

Työn nimi — Arbetets titel — Title

SPA-sovellukset hajautettuna järjestelmänä

Ohjaajat — Handledare — Supervisors

Dr. M. Luukkainen, Prof. T. Mikkonen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	May 3, 2020	65 pages

Tiivistelmä — Referat — Abstract

Single-page application (SPA) model has become a popular way of building web-applications. It makes the user experience of a website more similar to desktop-applications. This is achieved by not having to make a request to backend for each page navigation and operation.

However the SPA model brings some challenges of distributed data management to basic web-applications. Managing distributed consistency is a perennial research topic in computer science. Yet this has received little attention in single-page application context. This thesis compares single-page applications to distributed databases and aims to identify techniques from them that could be used in single-page applications.

The comparison begins by looking at different distributed databases and analysing which of them matches closest to the single-page application model. Then the comparison focuses on two topics. Techniques used by distributed databases for replicating data from site to another are presented and compared to techniques used in web application in communication between the server and the browser. Next topic of the thesis is to study how consistency related models like ACID-properties along with isolation levels and anomalies are realized or manifested in single-page applications.

As a result of the comparison it was observed that the data transfer methods used in distributed databases and web-applications were somewhat different from each other. Distributed systems favor push model for replication and replication is automatic from application developers perspective. Web applications ofter use pull-model and implementing replication is application developers responsibility. A set of consistency anomalies that can be manifested in single-page applications where found while analysing the consistency topic. The findings give a good starting point for developing libraries that could solve some of the problems that were found.

ACM Computing Classification System (CCS)
Information systems → Parallel and distributed DBMSs
Information systems → Web applications

Avainsanat — Nyckelord — Keywords

Single-page applications, distributed systems

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software Systems study track

Sisältö

1 Johdanto

Single-page application -mallista (SPA) on tullut erityisesti React.js JavaScript-kirjaston suosion myötä yleinen tapa totettaa verkkosivuja ja -sovelluksia. Perinteisten palvelimella luotujen HTML-näkymien sijaan SPA-mallissa näkymiä luodaan käyttäjän selaimessa JavaScript-kirjastojen avulla. Näkymien muodostamista varten palvelimelta ladataan selaimen muistiin dataa, jota sovellus näyttää käyttäjälle. Tätä samaa dataa säilytetään myös järjestelmän varsinaisessa tietokannassa palvelimella. Web-sovelluksen voidaan näin nähdä käyttävän hajautettua tietokantaa. Tällöin datan eheyden hallinnasta muodostuu haastava ongelma.

Tietokantoja ja hajautettuja järjestelmiä on tutkittu vuosikymmeniä ja eheyden hallintaan on kehitettu lukuisia menetelmiä ja malleja. SPA-sovellukset ovat sen sijaan tuoreempi ilmiö eikä niiden taustalla ole samanlaista vuosikymmenten akateemista tutkimusta. Tässä tutkielmassa vertaillaan SPA-sovelluksissa käytettyjä menetelmiä ja tekniikoita hajauttujen järjestelmien ja erityisesti hajautettujen tietokantojen käyttämiin datan replikoinnin ja datan eheyden hallinnan menetelmiin. Vertailun avulla pyritään vastaamaan seuraaviin tutkimuskysymyksiin:

- TK1: Millaisia hajauttujen tietokantojen piirteitä SPA-sovelluksilla on?

- TK2: Käytetäänkö SPA-sovelluksissa samankaltaisia datan replikointiin liittyviä menetelmiä kuin replikoiduissa tietokannoissa ja voisiko tietokantojen replikointimenetelmistä ottaa mallia SPA-sovelluksissa?

- TK3: Miten tietokantojen kontekstissa määritellyt erilaiset eheyshaasteet ja eheysmallit ilmenevät SPA-sovelluksissa?

Tutkielman rakenne on seuraavanlainen. Luvussa 2 käydään läpi SPA-malliin johtanutta web-teknologioiden kehitystä. Luvussa 3 esitellään mitä ovat SPA-sovellukset ja käydään läpi niiden motivaattoreita sekä toimintaperiaatetta. Luvussa 4 käsitellään hajautettujen tietokantojen määritelmiä ja syitä tietokantojen hajauttamiselle ja tutkitaan onko SPA-sovelluksilla samoja motivaattoreita. Näin tutkielman vertailu rajataan replikoituihin tietokantoihin, joiden perehdytään tarkemmnin luvussa 5. Luvussa 6 esitellään hajauttujen tietokantojen käyttämiä replikointiprotokollia eli tapoja datan synkronointiin tietokantapisteiden välillä. Näitä tapoja vertaillaan web-selaimen ja palvelimen välillä tapahtuvaan

kommunikaatioon käytettäviin tekniikoihin. Luvussa 7 käsitellään keskeistä hajautettuun dataan liittyvää haastetta eli eheyden hallintaa. Siihen liittyviä seikkoja peilataan SPA-sovellusten kontekstiin ja tutkitaan mitkä eheyden hallinnan osa-alueista olennaisia juuri SPA-sovelluksille. Luvussa 8 käydään läpi tutkielman tuloksia ja pohditaan mahdollisia seuraavia tutkimusaiheita. Lopuksi luvussa 9 tehdään vielä tiivis yhteenveto.

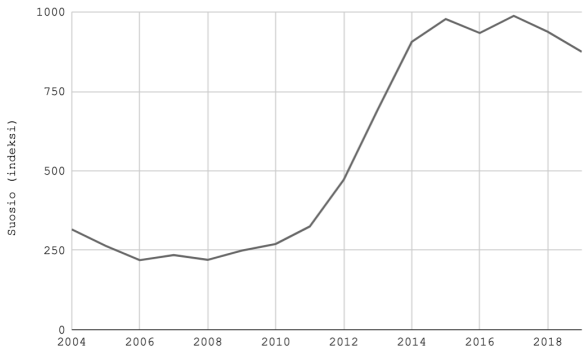
2 Web-tekniikoiden historiaa

Web-sovellusten kehittymisen ymmärtämiseksi on hyödyllistä tarkastella hieman historiaa. Tänä päivänä SPA-sovellukset ovat jopa niin suosittuja, että ne vaikuttavat oletusarvoiselle tavalle toteuttaa web-sovelluksia. On kuitenkin hyvä ymmärtää minkä ongelman ratkaisemiseksi SPA-malli on kehitetty. Historiaa on hyvä tarkastella myös web-selainten mahdollistamien ratkaisujen nykytilan hahmottamiseksi. Esimerkiksi tiedon tallentaminen paikallisesti ei ole aina ollut mahdollista selaimessa.

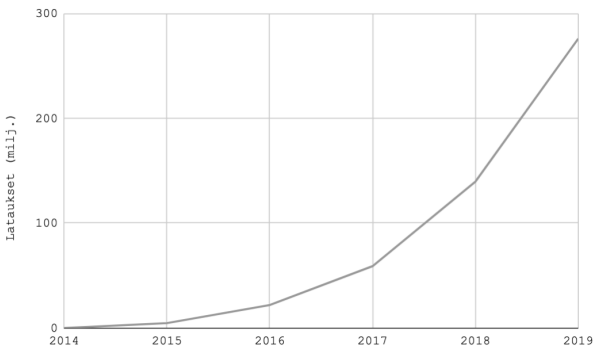
2.1 Ajax-pyynnöt

SPA-sovelluksille on olennaista mahdollisuus tehdä selaimesta asynkronisia pyyntöjä palvelimelle. Näistä pyynnöistä käytetään nimitystä Ajax, joka on lyhenne sanoista "Asynchronous JavaScript and XML". Termin lanseerasi Jesse James Garrett vuonna 2005 kirjoituksessaan *Ajax: A New Approach to Web Applications* [19]. Siinä hän kuvailee uuden tavan tehdä web-sovelluksia JavaScriptia ja asynkronisia XMLHttpRequest-pyyntöjä (XHR) käyttäen. Niiden avulla on mahdollista tehdä verkkosivuja, joissa sivua ei tarvitse ladata uudelleen jatkuvasti. Käyttökokemuksesta on näin mahdollista tehdä enemmän työpöytäsovellusten kaltainen. Vuonna 2006 julkaistu jQuery [18] helpotti Ajax-kyselyjen tekemistä ja selaimen esittämän HTML-näkymän dynaamista päivittämistä. JQeury-kirjasto muodostui seuraavia vuosina todella suosituksi ja sen avulla tehdyt web-sovellukset voidaan nähdä yhtenä ensiaskeleena staattisista HTML-sivuista kohti natiivisovelluksien kaltaisia SPA-sovelluksia.

Yksi mielenkiintoinen piirre Ajax-artikkelissa on se, että siinä käytetyt teknologiat olivat olleet selaimissa saatavilla jo useita vuosia. Malli olikin ollut jo käytössä esimerkiksi useissa Googlen palveluissa jo ennen artikkelin julkaisemista [19]. Uuden teknologian tai tekniikan esittelemisen sijaan kyseessä on pikemminkin menetelmän nimeäminen ja edistäminen, näyttäen samalla uutta suuntaan laajemmalle kehittäjäyhteisölle.



Kuva 2.1: Termin single-page application suosio Google-hauissa



Kuva 2.2: React.js -kirjaston latausmäärät npm-palvelusta

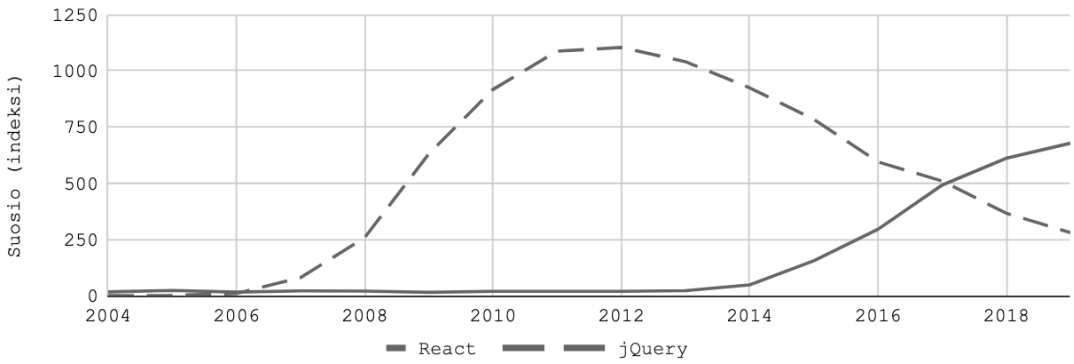
2.2 SPA-sovellukset

Termin single-page application alkuperä ei ole selkeä. Esimerkiksi kirja *Single Page Web Applications* [46] ei nimestään huolimatta mainitse termin alkuperää lainkaan. Wikipedia-artikkeli *Single-page application* on luotu vuonna 2008 ja siitä mainitaan, että termin olisi keksinyt Steve Yen vuonna 2005 [43]. Lähdeviittausta tälle ei kuitenkaan ole ja alkuperäistä blogikirjoitusta tai muuta sellaista ei ole helposti löydettävissä enää 15 vuotta myöhemmin.

Single-page application -malli on siis saanut alkunsa jossain vuoden 2005 tienoilla. Sen nouseminen suurempaan suosioon tapahtui kuitenkin vasta myöhemmin. Google Trends -palvelu mahdollistaa Googlen hakupalvelussa käytettyjen hakutermien suosion tutkimisen. Kuvassa 2.1 on esitetty historiatietoa termin “Single-page application” suosiosta. Siitä nähdään, että termin suosio on lähtenyt kasvuun vuosien 2010-2011 tienoilla. Suosion kasvua voi selittää vuonna 2010 Googlen julkaisema [47] AngularJS-ohjelmistokehys, joka toimi single-page application mallilla. Kuvassa 2.3 näkyy kuinka jQuery-kirjaston suosio Google-hauissa kääntyi laskuun pari vuotta AngularJS:n julkaisun jälkeen.

Vuonna 2013 ilmestyi tämän hetken suosituin SPA-mallia käyttävä kirjasto: React.js. Vuonna 2016 se oli Stack Overflow Developer Surveyn [51] “trending”-listan selvä voittaja. Sen suosio oli kasvanut edellisen vuoden kyselyyn verrattuna yli 300 prosenttia. Vuonna 2017 sen suosio ohittaa jQueryn kuvassa 2.3 ja vuonna 2019 sitä ladattiin peräti 275 miljoonaa kertaa npm-palvelusta. Kuvasta 2.2 nähdään kuinka React-kirjaston suosion kasvu on ollut huimaa.

SPA-mallin kehitys on toinen mielenkiintoinen esimerkki web-tekniikoiden kehittymisestä.



Kuva 2.3: Kirjastojen jQuery ja React suosio Google hauissa

Mallin mahdollistavat teknologiat (XHR-pyynnöt) ovat olleet selaimissa saatavilla jopa vuodesta 2000. Vuonna 2005 malli konkretisoitui siitä julkaistun artikkelin (Ajax) myötä. Muutamia vuosia myöhemmin mallia käyttäviä kirjastoja alkoi ilmestyä ja tekniikan suosio kasvoi. Tämän esimerkin perusteella näyttäisi siis, että uusien tekniikoiden kehittymisessä voi kestää yllättävän pitkään.

2.3 Datan tallentaminen selaimessa

Web-sivustojen kehittyessä on syntynyt tarve tallentaa dataa myös paikallisesti. HTTP-pyyntöjen mukana lähetettävät keksit (engl. cookie) mahdollistivat tämän, mutta parempia menetelmiä ryhdyttiin kehittelemään. Vuonna 2009 julkaistussa W3C luonnoksessa [29] esiteltiin suunnitelmat localStorage:sta, sessionStorage:sta sekä selaimessa toimivasta SQL-tietokannasta. Myöhemmin samana vuonna SQL-tietokannan osuus erotettiin omaan luonnokseensa [26]. Seuraavana vuonna W3C konsortio hylkäsi [27] ajatuksen SQL-tietokannasta ja sen tilalle syntyi luonnos [44] uudesta selaimassa toimivasta tietokannasta nimeltä Indexed Database. Vuonna 2013 localStoragen sisältävä Web Storage -esitys saavutti W3C Recommendation -tason. Tällä tasolla olevat spesifikaatiot mielletään standardeiksi, joita selainten oletetaan tukevan. Indexed Database saavutti tämän tason kaksi vuotta myöhemmin vuonna 2015.

Mahdollisuus datan paikalliseen tallentamiseen on olennainen tekijä Progressive Web App eli PWA-mallille. Sen esitteli Alex Russell kirjoituksessaan “Progressive Web Apps: Escaping Tabs Without Losing Our Soul” vuonna 2015 [55]. Russell listaa joukon ominaisuuksia tai vaatimuksia, joiden avulla web-sovelluksesta tehdään enemmän natiivisovellusten kaltaisia. Useimmat ominaisuuksista eivät ole relevantteja tämän tutkielman kannalta, mutta

vaatimus offline-käytettävyydestä on mielenkiintoinen, sillä se edellyttää datan tallentamista esimerkiksi Indexed Databasen avulla. Tällöin selain tallentaa pysyvästi paikallisen version tietokannan datasta.

Vuonna 2019 esitelty [39] *local-first software* -malli vie datan tallentamisen selaimessa vielä pidemmälle. Siinä datan pääasiallinen tallennuspaikka on selain, ja palvelimen tehtäväksi jää vain datan synkronointi eri käyttäjien ja laitteiden välillä. Verkkoyhteys palvelimelle ei ole pakollinen, vaan local-first -mallilla tehdyt sovellukset suunnitellaan niin, että kaikki operaatiot voidaan tehdä paikallisesti ja synkronoida myöhemmin palvelimelle.

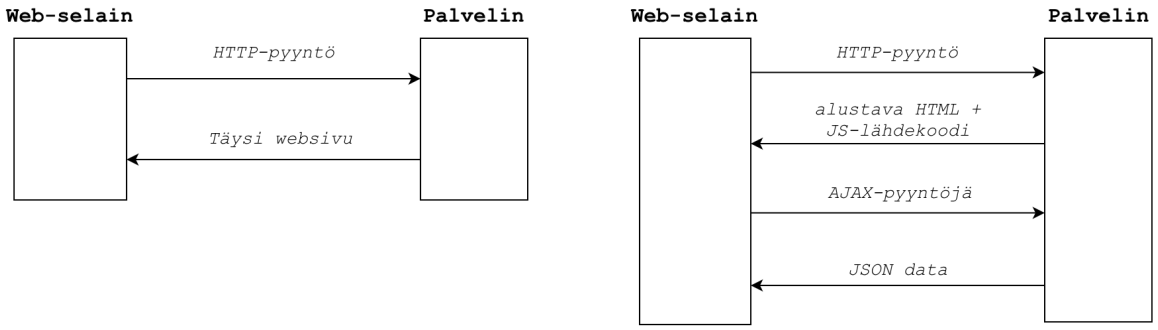
Taulukkoon 2.1 on kerätty aikajanalle tapahtumia, jotka auttavat ymmärtämään SPA-mallin ja selainten tallennusmekanismien nykytilaa. SPA-mallin syntyä edelsivät selainten XHR-tuki sekä Ajax-malli ja kirjastojen, kuten Angular ja React, myötä siitä on tullut todella suosittu tapa toteuttaa web-sovelluksia. Selainten tallennusmekanismit eivät suoranaisesti liity SPA-sovelluksiin, sillä ne eivät ole välttämättömiä SPA-sovelluksia varten. Tallennusmekanismit voivat kuitenkin olla olennaisia tulevia web-sovelluksien ohjelmointimallien syntyä ajatellen.

Vuosi	Tapahtuma
2000	**XMLHttpRequest** Asynkronisen kommunikaation selaimen ja palvelimen välillä mahdollistava XHR käytettävissä useissa selaimissa. [55]
2005	**Ajax** James Garrett konseptualisoi Ajax-käsitteen [19]
2006	**jQuery**-kirjasto julkaistiin [18]
2008	**Wikipedia: Single-page application** -artikkelin ensimmäinen versio [43]
2009	**Web Storage** -luonnos [29], jossa esiteltiin localStorage, sessionStorage ja WebSQL.
2010	**WebSQL ja IndexedDB** WebSQL hylättiin [27] ja tilalle [44] Indexed Database.
2010	**AngularJS** Google julkaisi vuonna 2010 AngularJS-ohjelmistokehyksen, joka toimi single-page application mallilla. [47]
2013	**WebStorage standardi** Web Storage (localstorage) W3C Recommendation [28].
2013	**React.js** -kirjaston julkaisu.
2015	**IndexedDB W3C Recommendation** Vuonna 2015 Indexed Database saavutti W3C Recommendation tason [45].
2015	**Progressive Web App** Termin ”Progressive Web App”lanseerasi Alex Russell kirjoituksessaan ”Progressive Web Apps: Escaping Tabs Without Losing Our Soul”[55].
2016	**React suosio** Vuoden 2016 Stack Overflow Developer Survey -kyselyssä [51] React-kirjaston suosio oli kasvanut huimat 311% edellisvuoteen verrattuna.

Taulukko 2.1: Aikajana web-tekniikoiden historiasta

3 SPA-sovellukset

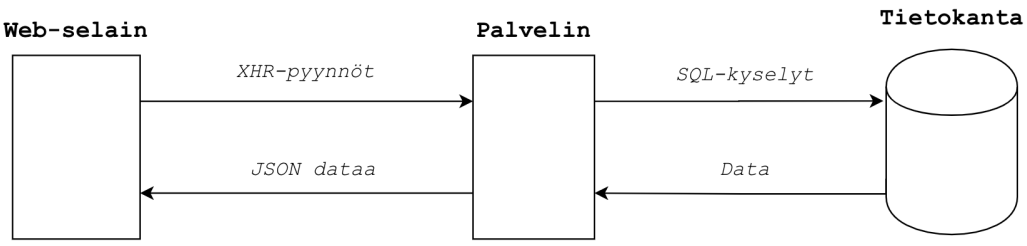
SPA-sovelluksiin on viitattu jo edellä useita kertoja, mutta on syytä esittää niille vielä tarkempi määritelmä. Single-page application -mallilla toteutetut verkkosivustot eli SPA-sovellukset ovat web-selaimella käytettäviä sovelluksia. Tavallisista verkkosivuista poiketen SPA-sovellusta varten ladataan vain yksi HTML-sivu. Tämän HTML-sivun lisäksi selaimeen ladataan palvelimelta sovelluksen tarvitsemat JavaScript- ja CSS-tiedostot sekä muita resursseja. Käyttäjän siirtyessä sovelluksessa toiselle sivulle ei tarvitakaan uuden HTML-sivun lataamista palvelimelta vaan uusi näkymä renderöidään selaimessa [15]. Kummankin mallin mukaista selaimen ja palvelin välillä tapahtuvaa kommunikaatiota on esitetty kuvassa 3.1.



Kuva 3.1: Perinteinen web-sovellus ja SPA-sovellus

Yksinkertaisen SPA-mallia käyttävä sovelluksen voidaan nähdä koostuvan kolmesta osasta. Kyseessä on kolmikerrosarkkitehtuuri ja sen osat on esitetty kuvassa 3.2. Käyttäjän laitteella on web-selaimessa toimiva JavaScript-sovellus, joka vastaa HTML-muodossa esitettävästä käyttöliittymästä. SPA-mallissa sovelluspalvelin tarjoaa tyypillisesti REST-rajapinnan, jonka kautta selainsovellus kommunikoi sovelluspalvelimen kanssa. Representational State Transfer (REST) on Roy Fieldingin [17] luoma arkkitehtuurityyli ohjelmointirajapintojen toteuttamiseen. REST ei ole sidottu mihinkään tiettyyn kommunikaatioprotokollaan, mutta tässä tutkielmassa oletetaan HTTP-pyyntöjä käyttävä REST-rajapinta. Kommunikaatio tapahtuu asiakkaan aloitteesta HTTP-pyynnöin ja palvelin vastaa pyyntöihin yleensä JSON-muodossa esitetyillä resursseilla.

SPA-sovellukset pyrkivät tekemään web-selaimessa toimivien sovelluksen käyttökokemuk-



Kuva 3.2: SPA-sovelluksen rakenne

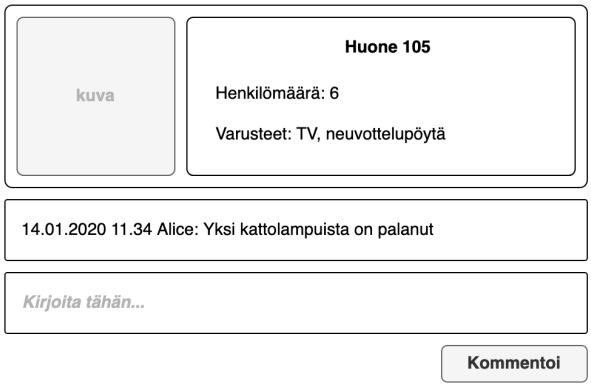
sesta yhtä sujuvaa ja responsiivista kuin työpöytäsovelluksissa [32]. Perinteiset verkkosivut koostuvat useista erillisistä HTML-sivuista, joiden välillä käyttäjä navigoi. Jokainen sivu ladataan erikseen palvelimelta ja tämä voi tehdä tällä mallilla toteutettujen sovellusten käyttämisestä tahmeaa. Jos sivuvaihdossa tai muussa interaktiossa oleva viive on alle 0.1 sekuntia, käyttäjä kokee järjestelmän reagoivan välittömästi. Tätä pidempikin vasteaika interaktioissa on hyväksyttävissä, mutta yli sekunnin viive koetaan häiritseväksi [49]. Perinteisissä verkkosovelluksissa sivulataukset asettuvat usein juuri 0.1 ja 1 sekunnin välille. Mikäli siis halutaan välittömästi reagoiva käyttökokemus on pyrittävä toteuttamaan sivuvaihdot ja muut interaktiot muulla tavoin. Single-page application -malli mahdollistaa työpöytäsovellusten kaltaisen kokemuksen web-selaimessa, sillä jokaista sivuvaihtoa varten ei jouduta tekemään pyyntöä palvelimelle vaan näkymät muodostetaan selaimessa paikallisesti etukäteen ladatun datan avulla.

3.1 React.js -käyttöliittymäkirjasto

React on vuonna 2013 julkaistu [50] JavaScript-kirjasto käyttöliittymien rakentamiseen. Se poikkeaa tavanomaisista HTML-templaatteja käyttävistä menetelmistä jakamalla käyttöliittymän osat komponentteihin [30]. Käyttöliittymänäkymän päivittäminen on tehty React:ssa hyvin helpoksi. HTML-elementtien imperatiivisten muokkausoperaatioiden kirjoittamisen sijaan sovelluskehittäjä päivittää käyttöliittymäkomponenttien käyttämää dataa eli tilaa ja React päivittää elementit automaattisesti. Tätä havainnollistetaan kuvassa 3.3 esimerkin avulla. Vasemmalla on esitetty JSON-objekti, joka kuvaa komponentin tilaa. Oikealla on piirros HTML-elementistä, jonka React-komponentti tuottaa. React-komponentit ovat siis kuin funktioita, jotka ottavat syötteenä tilaa kuvaavan objektin ja tuottavat HTML-elementtejä.

HTML-elementtien käsittelyn sijaan sovelluskehittäjä keskittyykin React-sovellusta ra-

```
{
  id: 'r-105',
  name: 'Huone 105',
  img: ''
  info: [
    { title: 'Henkilömäärä',
      value: 6 },
    { title: 'Varusteet',
      value: 'TV, neuvottelupöytä' }
  ],
  comments: [
    {
      id: '105-c-1',
      time: '2020-01-14 11.34.00',
      author: 'Alice',
      text: 'Yksi kattolampuista on palanut'
    }
  ]
}
```



Kuva 3.3: Tilaobjekti ja käyttöliittymäkomponentti

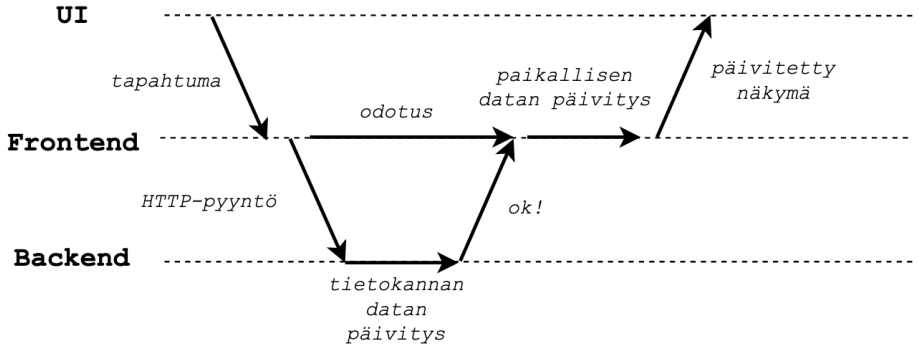
kentaessaan sovelluksen tilaan tehtäviin muutoksiin. React-sovelluksissa sovelluksen tilaa voidaan hallita useilla tavoilla ja paikoissa. Komponenteilla voi olla oma sisäinen tilansa tai sovelluksen juurikomponentti voi sisältää useiden komponenttien tarvitsemaa dataa. Tila voidaan säilyttää myös käyttöliittymäkomponteista erillään. Näin toimii Redux-kirjasto [2]. Siinä kaikki sovelluksen käyttämä tila voidaan keskittää yhteen paikkaan ja käyttöliittymäkomponentit tarkkailevat tarvitsemaansa osaa tilaobjektista.

3.2 Redux-tilanhallintakirjasto

Tässä tutkielmassa keskitytään tilanhallintamalliin, jossa sovelluksen tilaa hallitaan Redux-kirjaston avulla. Lisäksi oletetaan, että sovellukseen kuuluu sovelluspalvelin, joka tarjoaa REST-rajapinnan sekä relaatiotietokanta, kuten kuvassa 3.2. Lisäksi sovelluspalvelimeen on yhteydessä muita käyttäjiä omilla selaimillaan, mutta kuvassa esitetty vain yksi selain.

Redux-tilanhallintakirjastossa sovelluksen tilaa muuttavia operaatioita kutsutaan *actioneiksi*. Käyttöliittymän tapahtuman seurauksena voidaan kutsua *action creator* -funktiota, joka käynnistää actionin. Tämä käynnistetty action käsitellään *reducer*-funktiossa. Reducer sisältää kutakin actionia vastaavan sovelluksen tilaa muokkaavan funktion. Actionit voivat sisältää myös HTTP-pyyntöjä, joiden kautta operaatio suoritetaan myös palvelimella.

Perehdytään nyt sovelluksen tilan muuttamiseen Redux-kirjaston avulla. Kuvassa 3.4 on esitetty korkealla tasolla sovelluksen tilaa muuttavan operaation vaiheet, kun käytössä on React ja Redux selainpuolella ja tietokanta palvelinpuolella. Ensin operaatio tallennetaan palvelimelle ja onnistuneen vastauksen saatuaan selainsovellus suorittaa operaation myös paikalliseen tilaan. Paikalliseen tilaan tapahtunut muutos saa aikaan käyttöliittymän päivityksen ja käyttäjä näkee operaation onnistuneen.



Kuva 3.4: Redux-sovelluksen tietokantaoperaatio

Tarkastellaan samaa operaatiota vielä tarkemmin esimerkin avulla, sillä tilaan tehtävien muutoksien toteutus osoittautuu olennaiseksi detailiksi, kun tarkastellaan tilaa muokkaavien operaatioiden eli transaktioiden atomisuutta ja eristyneisyyttä aliluvussa 7.1.

Käyttäjän klikattua tapahtuman käynnistävää elementtiä, sovellus kutsuu tapahtumankäsittelijäfunktiota `commentItem`, joka on esitetty alla.

```
export const commentItem = (itemId, author, comment) => {
  return async dispatch => {
    const response = await itemService.addComment(itemId, author, comment)
    if (response.status === 200) {
      dispatch({
        type: 'NEW_COMMENT',
        data: { itemId: itemId, author: author, comment: comment },
      })
    }
  }
}
```

Funktio tekee ensin pyynnön palvelimelle ja sen jälkeen tallentaa muutoksen myös paikalliseen Redux-storessa säilytettävään tilaan, mikäli palvelinpyyntö onnistui. Operaatiot

Redux-storeen tehdään `dispatch`-funktion avulla. Sitä kutsuttaessa määritellään haluttu operaatio eli *action* sekä vaadittavat parametrit. Tässä esimerkissä operaatio on siis `NEW_COMMENT` ja sen tarvitsevat parametrina annetaan uusi kommenttiobjekti. Operaation toiminnallisuus määritellään reducer-funktiossa, joka näyttää seuraavanlaiselta:

```
const reducer = (state = {}, action) => {
  switch (action.type) {
    case 'NEW_COMMENT':
      return {
        ...state,
        items: items.map(item => {
          item.id !== action.data.item
            ? return item
            : return {
                ...item,
                comments: [...item.comments, action.data.comment]}})}
    default:
      return state
}}
```

Tarkastellaan hieman tarkemmin tämän `reducer`-funktion toteutusta. Todellisuudessa `reducer`-funktion `switch`-lauseke sisältää useita kohtia, mutta ne on jätetty tässä esittämättä. Jokaista määriteltyä action:ia vastaa yksi `switch`-lausekkeen kohta, jossa on ohjelmoitu operaation tekemät muutokset Redux-store:n tilaan. Operaatio on tehtävä edellistä tilaa mutatoimatta. Tähän käytetään esimerkissä JavaScriptin *object spread* ja *array spread* -operaattoreita. Ne ovat käytännöllisiä reducer-funktioiden toteuttamiseen, sillä ne eivät mutatoi käsiteltävää tietorakennetta vaan palauttavat uuden objektin. Tässä operaatio käy läpi `state.items`-listan ja lisää kommentin oikeaan item-objektiin.

Poimitaan tästä melko yksityiskohtaisesta selostuksesta vielä muutama oleellinen yksityiskohta esille. Funktiosta `commentItem` nähdään, että operaatio tehdään ensin palvelimelle ja sitten paikalliseen tilaan. Nämä voitaisiin kuitenkin tehdä myös päinvastaisessa järjestyksessä, eli paikallinen tila voidaan päivittää jo ennen kuin palvelin on vastannut operaation onnistuneen. Näin voidaan siis valita synkronisuuden ja asynkronisuuden väliltä operaatiokohtaisesti. Lisäksi tilaa päivittävän `reducer`-funktion toiminta on olennainen. Siinä ei muokata olemassa olevaa tilaa vaan luodaan uusi objekti. Lisäksi selaimen yksisäikeisyyden ansioista vain yksi reducer-operaatiot suoritetaan selkeässä peräkkäisjärjestyksessä ilman mahdollisuutta rinnakkaisuudesta johtuviin ongelmiin.

4 Hajautetut tietokannat

Tässä luvussa pyritään vastaamaan tutkielman ensimmäiseen tutkimuskysymykseen eli selvitetään millaisia hajautettujen tietokantojen piirteitä SPA-sovelluksilla on. Tätä varten perehdytään ensin ominaisuuksiin, joiden avulla määritellään hajauttuja tietokantoja ja analysoidaan toteutuvatko nämä ominaisuudet myös SPA-sovelluksissa. Tämän lisäksi perehdytään vastaavasti hajauttujen tietokantojen tavoittelemiin ominaisuuksiin ja vertaillaan onko SPA-sovelluksilla samoja tavoitteita.

4.1 Määrittelevät ominaisuudet

On haasteellista antaa hajautetuille tietokannoille suppea ja täsmällinen määritelmä. Sen sijaan on helpompaa kuvailla niitä ominaisuuksia mitä hajautetuilla tietokannoilla tyypillisesti on. Kirjassa Database System Concepts [40] käytetään tällaista määrittelytapaa. Seuraavassa käydään läpi kirjassa listatut ominaisuudet ja arvioidaan miten ne toteutuvat SPA-sovellusten kohdalla.

Tietokanta sijaitsee usealla tietokoneella. Tämän kohdan tulkinta on tutkielman kannalta olennainen. Jos SPA-sovelluksen paikallinen tila esimerkiksi Reduxissa ajatellaan olevan osa järjestelmän tietokantaa, niin tällöin tietokanta sijaitsee usealla tietokoneella. Paikallinen tila voitaisiin nähdä kenties myös varsinaisesta tietokannasta erillisenä välimuistina eikä varsinaisena osana tietokantaa. Tässä tutkielmassa paikallinen tila nähdään olennaisena osana järjestelmän tietokantaa, jolloin kyseessä on hajautettu tietokanta.

Pisteiden (engl. site tai node) **välinen kommunikaatio tapahtuu verkon yli.** SPA-sovelluksessa tietokannan pisteet sijaitsevat käyttäjän selaimessa ja palvelimella. Niiden välinen kommunikaatio tapahtuu verkon yli, joten tältä osin SPA-sovellukset muistuttavat hajautettuja tietokantoja.

Pisteillä ei ole jaettua muistia tai levyä. SPA-sovelluksessa tietokantapisteet sijaitsevat täysin erillisillä laitteilla, joten niillä ei ole jaettua muistia.

Laitteistot voivat olla erilaiset tietokannan eri pisteissä. Osa SPA-sovelluksesta sijaitsee käyttäjän laitteella ja osa palvelimella. Nämä ovat tavallisesti erilaiset laitteistol-

taan. Varsinkin käyttäjän laitteet voivat olla hyvinkin vaihtelevia, sillä SPA-sovelluksia voidaan käyttää niin tavallisilla tietokoneilla kuin tableteilla ja puhelimilla.

Pisteet on **tyypillisesti maantieteellisesti hajautettu.** Käyttäjän laite ja sovelluspalvelin voivat sijaita hyvinkin etäällä toisistaan, joten SPA-sovellus on yleensä maantieteellisesti hajautettu.

Transaktio voi lukea ja käsitellä dataa toisesta pisteestä kuin omasta aloituspisteestään. SPA-sovelluksessa transaktion aloituspisteeksi voidaan ajatella käyttäjän webselain. Käyttäjän tekee operaatioita, jotka muokkaavat dataa sekä paikallisessa Redux-tietokannassa että HTTP-pyynnön kautta myös palvelimella sijaitsevassa tietokannassa.

4.2 Tavoiteltavat ominaisuudet

Tietojärjestelmän tai tietokannan hajauttamiselle sekä datan replikoinnille on monia syitä ja listaukset vaihtelevat hieman lähteestä ja näkökulmasta riippuen. Seuraavassa on koostettu listaus eri lähteissä mainituista motivaattoreista koskien niin hajautettuja järjestelmiä yleensä kuin hajautettuja tietokantoja. Näin muodostetaan ensin laaja käsitys järjestelmien hajauttamisen syistä ja arvioitua mitkä näistä ovat olennaisia myös SPA-sovellusten kohdalla. Lisäksi saadaan muodostettua käsitys myös siitä millaisiin hajautettuihin tietokantoihin SPA-sovelluksia kannattaa vertailla.

Maantieteellisesti hajautettu ympäristö voi olla motivaatio tai syy järjestelmän hajauttamiselle [20]. Järjestelmään kuuluessa olennaisesti toisistaan fyysesti erillään olevia laitteita, kuten pankkiautomaatteja, järjestelmä on väistämättä hajautettu. Kyseessä voi siis olla enemmänkin toimintaympäristön asettama rajoite kuin tavoiteltava ominaisuus. Hajauttamalla tietokanta maatieteellisesti voidaan saavuttaa muita haluttuja ominaisuuksia. SPA-sovellukset toimivat yleensä maantieteellisesti hajautussa ympäristössä: asiakasohjelma eli selainsovellus sijaitsee käyttäjän omalla laitteella ja sovelluspalvelin sekä tietokanta konesalissa kaukana käyttäjästä.

Laskentanopeus, joka on mahdollista saavuttaa yhdellä prosessorilla on rajallista ja joissain tapauksissa riittämätöntä. Yksittäisen operaation vaatimaa laskentaa täytyy siis jakaa useampaan osaan, joita voidaan laskea rinnakaisesti useilla prosessoreilla [20]. Näin toimivat **rinnakkaistietokannat** (engl. parallel database). SPA-sovelluksissa ei ole ole kyse laskennan hajauttamisesta tällä tavalla.

Skaalautuvuus [57] tarkoittaa hajautetuissa järjestelmissä monia asioita. Se voi tarkoit-

taa esimerkiksi kuormantasausta ja useiden pyyntöjen palvelemista. Tämä on relavantti piirre myös SPA-sovelluksissa. Järjestelmä pystyy tasaamaan lukuoperaatioiden kuormaa selaimessa sijaitsevan datan avulla. Käyttäjän tehdessä sivustolla operaation, joka vaatisi tavanomaisessa toteutustavassa lukuoperaatiota tietokantaan, voidaankin SPA-sovelluksessa lukea tarvittava data selaimen muistista. Jos lukuoperaatiot ovat raskaita, kuten hakuoperaatioita, tästä voi olla merkittävä etu järjestelmän suorituskyvylle. Skaalautuvuus voi tarkoittaa myös suuren datamäärän hallitsemista useiden palvelimien avulla [38]. Tällöin tietokannan data jaetaan useiden palvelimien kesken siten, että yksittäinen palvelin säilyttää vain tiettyä osaa datasta. Tällaisia ovat **partitioidut tietokannat**. SPA-sovelluksissa taustajärjestelmän tietokanta huolehtii tyypillisesti koko datamäärästä ja selainten paikallinen data on lähinnä väliaikainen kopio tämän datan osasta.

Datan lukemiseen liittyvää **vasteaikaa voidaan pienentää** viemällä kopio datasta lähellä asiakasta [34, 38]. Näin toimivat varsinkin **replikoidut tietokannat**. Tämä on tutkielman näkökulmasta kenties olennaisin syy tietokannan hajauttamiselle. SPA-sovelluksien tärkeä motivaattori on työpöytäsovellusten kaltaisen responsiivisen käyttökokemuksen tuominen selaimeen. Kopioimalla dataa useisiin pisteisiin voidaan viedä data lähemmäs asiakkaita ja näin pienentää datan lukemisen vasteaikaa. SPA-sovelluksissa data viedään mahdollisimman lähelle asiakasta: hänen omalle laitteelleen selaimen muistiin.

Resurssien jakaminen eri prosessien tai käyttäjien kesken voi olla syy järjestelmän hajauttamiselle. Ghosh [20] jakaa jaettavat resurssit kahteen kategoriaan: laitteistoresursseihin ja ohjelmistoresursseihin. Jaettavia laitteistoja voivat olla esimerkiksi printterit tai kovalevyt. Jaettavasta ohjelmistoresurssista esimerkkinä käytetään Google Docs tekstinkäsittelysovelluksen käyttämistä selaimen kautta. Tällöin käyttäjä ei itse asenna käyttämäänsä sovellusta vaan hyödyntää palvelimelle asennettua ohjelmistoa. Nämä esimerkit jättävät kuvan jaettavista resursseista hieman kapeaksi, mutta Silberschatz [40] listaa edellisten lisäksi myös kaksi tämän tutkielman näkökulmasta olennaista resurssia: data ja tiedostot. Esimerkkinä käytetään yliopistojen yhteistä hajauttua tietokantaa. Ilman yhteistä järjestelmää tiedonvaihto jouduttaisiin tekemään muulla tavalla. Myös SPA-sovelluksissa jaetaan ohjelmistoresursseja käyttäjien kesken, sillä käyttäjät lataavat sovelluksen selaimeensa eikä erillistä asentamista vaadita. Myös palvelimen tietokannan tarjoama tallennuskapasiteetti voidaan nähdä resurssien jakamisena.

Hajauttamalla tietokanta on mahdollista parantaa järjestelmän **vikasietoisuutta ja saatavuutta**. Yhdellä palvelimella suoritettavat ohjelmat ovat alttiita esimerkiksi laitteisto-

vioista johtuville häiriöille ja katkoksille. Hajautetut järjestelmät pystyvät selviämään vastaavista vioista erilaisten häiriönsietomenetelmien avulla [20, 40, 57, 34]. Replikoidut tietokannat parantavat datan saatavuutta tarjoamalla useita pisteitä, joista sovellukset voivat lukea dataa. Tällöin yhden palvelimen vikaantuessa pyyntöjä voidaan palvella toiselta palvelimelta [20]. SPA-malli parantaa selainsovellusten ja -sivustojen vikasietoisuutta, sillä sivuston selailuun tarvittava data voidaan ladata valmiiksi laitteelle. SPA-malli parantaa siis saatavuutta käyttäjille, jotka ovat jo aloittaneet sovelluksen käytön. Uudet käyttäjät eivät voi aloittaa sovelluksen käyttöä tietokantapalvelimen ollessa saavuttamattomissa. Uudet käyttäjät eivät myöskään pysty hyödyntämään muiden käyttäjien paikallisia kopioita datasta. SPA-malli parantaa siis saatavuutta samaan tapaan kuin replikoidut tietokannat, mutta vain olemassa oleville käyttäjille.

Paikallinen autonomia on Silberschatzin [40] mukaan yksi hajauttujen tietokantojen merkittävimmistä eduista. Voidaan siis jakaa esimerkiksi datan omistajuus tietokannan pisteiden kesken niin, että kukin piste hallitsee itselleen kuuluvaa dataa sen sijaan, että yksi keskuspiste hallinnoisi kaikkea dataa. Tästä voidaan käyttää esimerkkinä vähittäistavarakauppaa, jossa kullakin kaupalla on oma tietokantapisteensä. Jokainen kauppa on vastuussa esimerkiksi oman varastosaldonsa ylläpitämisestä. Paikallinen autonomia liittyy myös toimintavarmuuteen [14]. Esimerkiksi kauppa haluaa ylläpitää varastosaldoaan vaikka järjestelmässä olisi häiriö, joka estää kommunikaation muihin pisteisiin. Paikallinen autonomian tavoitteleminen ei ole tunnusomaista SPA-sovelluksille, mutta yleistyessään local first -malli [39] voi viedä SPA-sovelluksia tähän suuntaan.

Taulukkoon 4.1 on koottu tiiviiseen muotoon edellä tehnyt analyysit. Hajautettujen tietokantojen tavoittelemat ominaisuudet on kerätty riveille ja sarakkeissa ovat kolme esiteltyä erityyppistä hajautettua tietokantaa sekä viimeiseen sarakkeeseen SPA-sovellus.

	Tietokantatyyppi			
Tavoiteltava ominaisuus	**Rinnakkais-laskenta**	**Partitioitu**	**Replikoitu**	**SPA**
Maantieteellinen hajautus		X	X	X
Laskentanopeus	X	X		
Kuormantasaus			X	(X)
Suuren datamäärän käsittely	X	X		
Pieni vasteaika			X	X
Resurssien jakaminen		X	X	(X)
Vikasietoisuus			X	X
Paikallinen autonomia			X	(X)

Taulukko 4.1: Hajautettujen tietokantojen motivaattorit

Tästä hajautettujen järjestelmien ja tietokantojen motivaattoreiden listauksesta huomataan, että useat niistä ovat relavantteja myös SPA-sovellusten kohdalla. Osa motivaattoreista pätee jossain määrin SPA-kontekstissa, mutta selkein yhtymäkohta on vasteajan pienentäminen. Se on olennainen syy käyttää SPA-mallia, sillä juuri pieni vasteaika erottaa SPA-sovellukset perinteisen mallin web-sovelluksista. Tämän havainnon perusteella kannattaa kiinnittää huomioita hajautettuihin tietokantoihin, jotka pyrkivät erityisesti pienentämään vasteaikaa. Tällaisia ovat replikoidut tietokannat.

5 Replikoidut tietokannat

Tässä luvussa perehdytään tarkemmin replikoituihin tietokantoihin. Replikoituja tietokantoja voidaan luokitella ainakin niiden käyttämien isäntäpisteiden (engl. master) määrän, topologioiden, replikoinnin synkronisuuden sekä pisteiden homogeenisuuden suhteen. Käydään läpi näitä tekijöitä ja arvioidaan SPA-sovelluksia, kunkin osa-alueen näkökulmasta. Tavoitteena tarkentaa vielä näkemystä siitä, että millaisia replikoituja tietokantoja SPA-mallin asetelma muistuttaa eniten.

5.1 Arkkitehtuurit

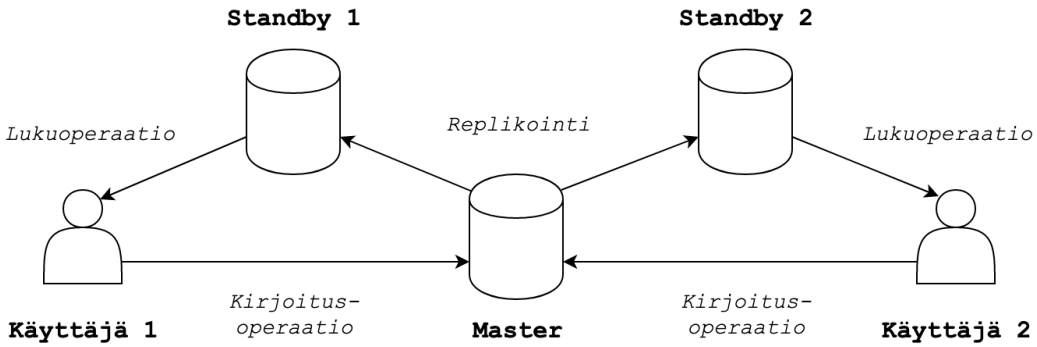
Replikoidut tietokannat voidaan jakaa kolmeen kategoriaan sen mukaan kuinka monta isäntäpistettä eli kirjoitusoperaatioita sallivaa pistettä järjestelmässä on. Kategoriat ovat [38]:

- **Master-standby** -arkkitehtuuri eli yhden isäntäpisteen malli.
- **Multi-master** -arkkitehtuuri eli useiden isäntäpisteiden malli.
- **Leaderless**-arkkitehtuuri eli yksikään pisteistä ei toimi operaatioita koordinoivana isäntänä.

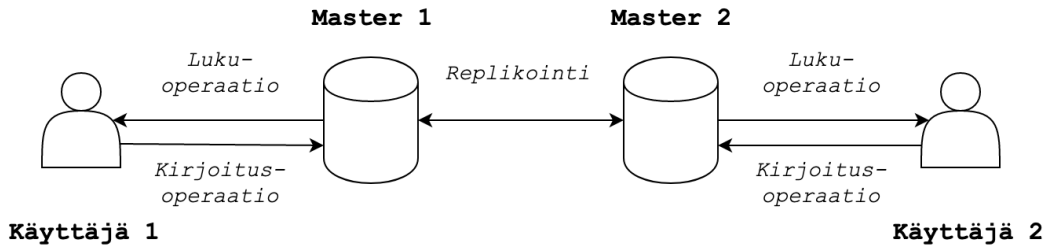
Usein isäntäpisteitä on yksi eli järjestelmä noudattaa master-standby -arkkitehtuuria [38], kuten kuvassa 5.1. Siinä kaikki kirjoitusoperaatiot lähetetään keskitetylle isäntäpisteelle eli master-palvelimelle. Se suorittaa operaation paikallisesti ja lähettää muutoksen muihin pisteisiin, joita kutsutaan standby-palvelimiksi. Lukuoperaatiota tehdessään asiakkaat voivat käyttää standby- tai master-palvelinta.

Isäntäpisteitä voi olla myös useita, jolloin kyseessä on multi-master -arkkitehtuuri [38], kuten kuvassa 5.2. Kaikki järjestelmän pisteet voivat siis toimia isäntäpisteinä ja vastaanottaa kirjoitusoperaatioita asiakasohjelmilta.

Replikoitu tietokanta voi olla toteutettu myös niin, että yksikään pisteistä ei toimi isäntänä. Tätä kutsutaan leaderless-arkkitehtuuriksi [38]. Tilanne muistuttaa multi-master -arkkitehtuuria, sillä kaikki pisteet voivat vastaanottaa kirjoitusoperaatioita. Eroavaisuus syntyy siitä, että asiakasohjelmistot voivat suorittaa operaatioita kommunikoimalla usei-

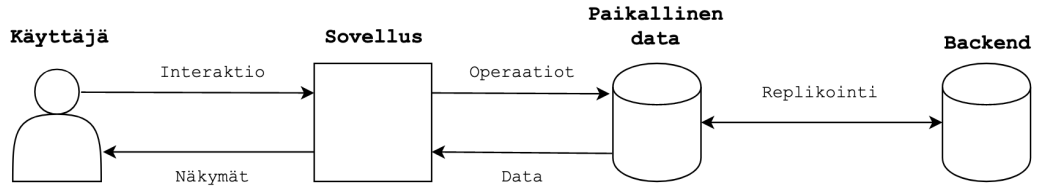


Kuva 5.1: Master-standby -arkkitehtuuri



Kuva 5.2: Multi-master -arkkitehtuuri

den pisteiden kanssa. Päivitysoperaatiota varten asiakasohjelma tekee pyynnön yli puolelle tietokannan pisteistä. Asiakkaalle riittää tietokantapisteiden enemmistön hyväksynnät, jotta se voi varmistua operaation onnistumisesta. Leaderless-arkkitehtuurissa koordinointivastuuta on siis siirretty asiakkaille, kun yksikään tietokantapisteistä ei toimi isäntänä ja operaatioden koordinaattorina.

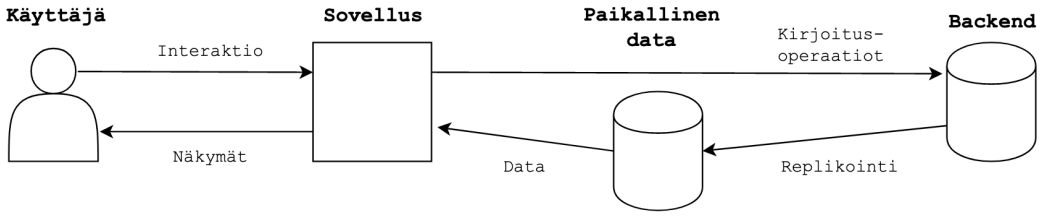


Kuva 5.3: SPA-sovelluksen arkkitehtuuri

Kuvassa 5.3 on esitetty SPA-sovellus niin, että vertailu replikoituihin tietokantoihin on helppoa. Siinä käyttäjä näkee sovelluksen tuottamia näkymiä ja on vuorovaikutuksessa sovelluksen kanssa sen käyttöliittymän kautta. Sovellus muuntaa käyttäjän interaktiot operaatioiksi, jotka muuttavat sovelluksen tilaa. Sovellus suorittaa operaatiot funktiokut-

suina paikalliseen tilanhallintajärjestelmään, joka välittää ne edelleen palvelimelle ja tietokantaan. Jos palvelin ei sisällä erityistä logiikkaa, voidaan tämä paikallisen tilanhallinnan ja palvelimella sijaitsevan tietokannan välinen operointi nähdä kahden tietokantapisteen välisenä replikointina tai osittaisena replikointina.

SPA-sovellukset ovat selvästi lähempänä multi-master kuin master-standby -arkkitehtuuria, jos tarkastellaan kuvia 5.1, 5.2 sekä kuvaa 5.3. Multi-master -arkkitehtuurissa asiakas kommunikoi yhden tietokantapisteen kanssa ja tietokantajärjestelmä hoitaa päivitysten propagoinnin muihin pisteisiin [40]. Kuvassa 5.3 SPA-sovelluksen toiminta on esitetty juuri tällä tavalla.



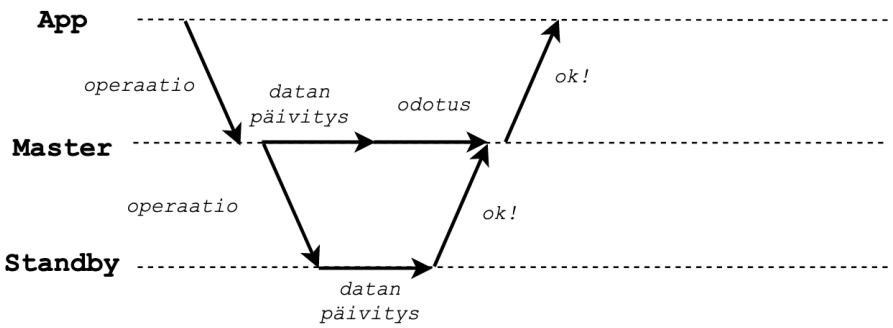
Kuva 5.4: SPA-sovelluksen master-standby versio

SPA-sovellus olisi kuitenkin mahdollista rakentaa ja kuvata myös enemmän master-standby -arkkkitehtuuria mukaillen. Kuvassa 5.4 on esitetty tätä lähestymistapaa. Asiakassovelluksella on oma kopio datasta paikallisessa muistissaan, kuten aiemminkin. Tässä tapauksessa paikallinen kopio on tosin ainoastaan luettavissa. Kaikki päivitysoperaatiot on tehtävä suoraan master-pisteenä toimivalle palvelimelle. Palvelin puolestaan hoitaa päivitysten propagoinnin asiakkaille, joiden paikallinen kopio datasta vastaa master-standby -arkkitehtuurin standby-pistettä.

5.2 Synkroninen vai asynkroninen replikointi

Replikoidun tietokannan toiminnassa on olennaista tehdäänkö replikointi synkronisesti vai asynkronisesti. Jotta kaikki tietokannan pisteet olisivat jatkuvasti samansisältöisiä eli keskenään eheitä on replikointi tehtävä synkronisesti. Tämä kuitenkin voi aiheuttaa suorituskykyhaasteita, joten replikointi voidaan eheyden vaarantumisesta huolimatta päätyä tekemään asynkronisesti. Tämä ongelma on kuvailtu lukuisissa hajauttuja tietokantoja käsittelevissä lähteissä [23, 57, 20, 40, 38].

Synkroninen replikointi tarkoittaa sitä, että tietokannan piste vastaa asiakkaalle ope-

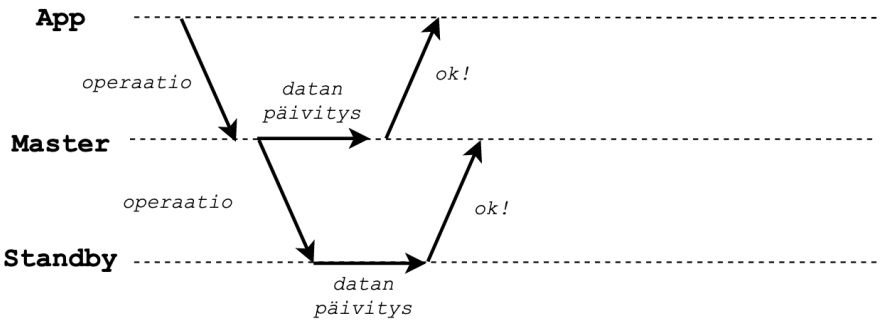


Kuva 5.5: Synkroninen replikointi

raation onnistuneen vasta, kun päivitys on saatu toteutettua jokaisella hajautetun tietokannan pisteistä. Yksinkertaistettu versio prosessista on esitetty kuvassa 5.5. Tämä operaatioiden globaali atomisuus toteutetaan usein kaksi- tai kolmevaiheisella sitoutumiskäytännöllä. Siinä transaktion aloittanut piste kysyy ennen operaation sitoutumista kaikilta muilta tietokannan pisteiltä, että ovatko ne valmiita sitouttamaan operaation. Jos kaikki vastaavat myöntävästi, aloittaja vastaa myös asiakkaalle hyväksyvästi. Jos yksikin vastaa kieltävästi transaktio perutaan. Samoin, jos kaikilta pisteiltä ei saada vastausta, transaktio joudutaan perumaan. Tämä onkin merkittävä haaste synkronisessa replikoinnissa. Tietokantapisteiden välisen verkkoyhteyden häiriöt estävät päivitysten tekemisen kokonaan. Järjestelmän saatavuus voi siis kärsiä synkronisuudesta. Lisäksi synkronisuuden vaatima järjestelmän laajuinen koordinointi vaikuttaa järjestelmän suorituskykyyn merkittävästi myös tietoliikenteen toimiessa normaalisti.

Useissa tapauksissa ainoaksi ratkaisuksi jääkin eheysvaatimusten laskeminen sallimalla replikoinnin asynkronisuus. Tällöin tietokantapiste voi vastata asiakkaalle operaation onnistumisessa ennen vahvistuksen kysymistä muilta tietokantapisteiltä. Tätä havainnollistetaan kuvassa 5.6. Menettelystä käytetään myös nimitystä laiska replikointi (engl. lazy replication) [23]. Seurauksena on kuitenkin mahdollisia ongelmia datan eheyden kanssa, sillä tietokantapisteet ovat vähintäänkin hetkittäin eri sisältöiset.

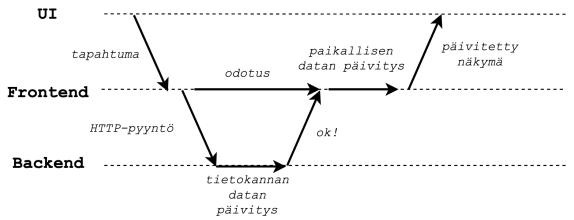
Eric Brewer kiteytti tämän eheyden ja saatavuuden välisen valintatilanteen CAP-teoreemaksi vuonna 2000 [11]. Teoreema on saanut nimensä sanoista *consistency*, *availability* ja *partition tolerance* eli suomeksi eheys, saatavuus ja osituksen sietokyky. Näistä viimeinen tarkoittaa siis käytännössä järjestelmän sisäisten verkkokatkosten sietokykyä. Sana *partition* viittaa tässä siihen, että hajauttu järjestelmä on jakautunut osiin, jotka eivät kykene kommikoimaan keskenään. Teoreeman väite on siis se, että hajautettu järjestelmä kykenee toteuttamaan korkeintaan kaksi näistä mainituista ominaisuuksista.



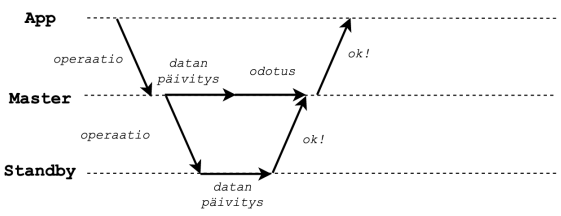
Kuva 5.6: Asynkroninen replikointi

Verkkoyhteyden häiriöiden eli järjestelmän osituksen sattuessa on täten valittava eheyden ja saatavuuden väliltä. Asynkroninen replikointi tarkoittaa saatavuuden valitsemista ja vastaavasti synkroninen eheyden valitsemista.

Jos tarkastellaan luvussa 3 esiteltyä SPA-sovelluksen toimintaa niin voidaan todeta, että siinä on kyseessä synkronisesta replikoinnista mikäli tarkastellaan palvelinta ja vain yhden käyttäjän laitetta. Päivitys paikalliseen tilaan tehdään vasta, kun palvelin on vastannut operaation onnistuneen tietokannassa. Synkroninen replikointi kahden tietokantapisteen välillä toimii vastaavasti. Tilanne kuitenkin puuttuu, jos tarkastellaan kaikkien käyttäjien laitteita. Synkroninen replikointi kaikkiin laitteisiin vaatisi päivityksen lähettämistä palvelimelta muiden käyttäjien selaimiin tai peer-to-peer -yhteyksien käyttämistä päivityksen levittämisessä. Tämä ei kuitenkaan ole helposti toteutettavissa käyttämällä ainoastaan HTTP-pyyntöjä käyttävää REST-rajapintaa selaimen ja palvelimen väliseen kommunikaatioon. Lisäksi synkroninen replikointi lukuisien muiden käyttäjien laitteisiin mahdollisesti epävarmojen verkkoyhteyksien yli olisi melkoisen haastava tehtävä ja sitä voitaneen pitää harvinaisena erikoistapauksena. Näin synkronista replikointia on on käytännöllistä tarkastella yhden käyttäjän ja palvelimen välillä.



Kuva 3.4



Kuva 5.5

Kuva 5.7: Operaatioiden vertailu

Vertailun helpottamiseksi kuvassa 5.7 on esitetty rinnakkain aiemmin esitellyt SPA-sovelluksen tietokantaoperaation vaiheita sekä synkronisen replikoinnin vaiheita esittävät kuvat. Näissä eroavaisuutena on käytännössä vain keskimmäisen kerroksen toiminta. SPA-sovelluksessa paikallinen tila päivitetään vasta palvelimelta saadun vastauksen jälkeen. Hajautetuissa tietokannoissa päivitys voidaan tehdä isäntäpisteellä jo ennen toisen pisteen vahvistusta, mutta se sitoutetaan vasta vahvistuksen jälkeen.

SPA-sovelluksissa voidaan kuitenkin tehdä replikointi myös asynkronisesti. Tätä käytetään esimerkiksi, kun halutaan käyttää mallia, josta käytetään nimityksiä optimistiset päivitykset tai optimistinen käyttöliittymä (engl. optimistic UI) [9]. Siinä käyttöliittymä päivitetään heti käyttäjän tehtyä jonkin operaation ilman, että vastausta operaation onnistumisesta odotetaan palvelimelta.

5.3 Varioituja arkkitehtuureja

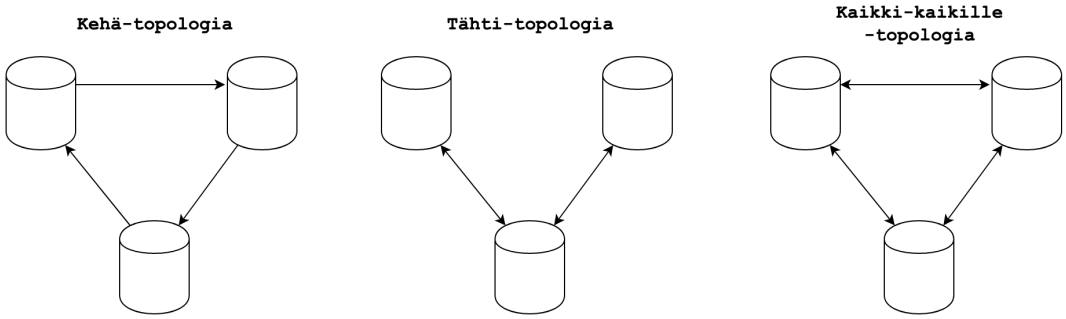
PostgreSQL käyttää dokumentaatiossaan 5.1 kuvan kaltaisesta arkkitehtuurista nimitystä *hot standby*. Sen lisäksi on olemassa niin kutsuttu *warm standby* -arkkitehtuuri, jossa standby-palvelimet eivät salli yhteyksiä asiakasohjelmilta. Ne toimivat ainoastaan ajantasaisina varmuuskopioina ja varajärjestelmänä häiriötilanteita varten. Mikäli master-palvelin lakkaa toimimasta, voidaan ottaa käyttöön warm standby -palvelin. SPA-sovelluksiin sovellettuna warm standby -malli voisi tarkoittaa paikallisen datan käyttämistä vain, jos verkkoyhteyspalvelimelle on poikki tai hidas. Muutoin sovellus lataisi jokaisen näkymän palvelimelta esimerkiksi varmistaakseen datan ajantasaisuuden.

Master-standby -mallista on olemassa [12, 57] myös variantti, jossa tietokantapisteet voivat vaihtaa rooleja. Näin voidaan väliaikaisesti siirtää datan kirjoitusoikeus jollekin standby-palvelimista. Tämä voi tehostaa useiden peräkkäisten päivitysoperaatioden suorittamista, kun kirjoitukset sallitaan lähellä päivityksiä tekevää asiakasta.

Tämän mallin mainitaan soveltuvan myös offline-tilassa toimivien mobiililaitteiden (mobile computers) käyttöön. Ennen verkkoyhteyden katkaisemista mobiililaite pyytää itselleen oikeuden toimia tietokannan isäntäpisteenä. Näin laite voi tehdä päivityksiä dataan verkkoyhteyden puuttumisesta huolimatta ja palatessaan verkkoon päivitykset lähetään muihin pisteisiin ja master-rooli siirretään takaisin normaalille palvelimelle. Tämä käyttötapa on suoraan sovellettavissa SPA-sovelluksiin, joita käytetään usein juuri mobiililaitteilta.

Kun multi-master -arkkitehtuuria käyttävä tietokanta sisältää vähintään kolme

isäntäpistettä, on mahdollista järjestää replikointi käyttäen erilaisia topologioita [38]. Kuvassa 5.8 on esitetty kolme esimerkkiä. Kehä-topologiassa (engl. circular) jokainen piste huolehtii päivitysten propagoinnista ainoastaan yhdelle muista isäntäpisteistä. Tähti-topologiassa (engl. star) yksi pisteistä on keskeisessä roolissa ja kaikki muut pisteet lähettävät päivityksensä sille. Keskuspiste propagoi saamansa päivitykset edelleen kaikille muille pisteille. Kaikki-kaikille topologiassa kukin piste lähettää päivitykset suoraan kaikkiin muihin pisteisiin.



Kuva 5.8: Multi-master -topologiat

Näistä topologioista tähti-topologia vastaa parhaiten SPA-sovelluksia. Siinä yksi pisteistä on keskeisessä asemassa ja kaikki päivitykset kulkevat sen kautta. SPA-sovelluksissa tämä keskeinen piste on palvelin. Kehä- ja kaikki-kaikille -topologiat vaatisivat suoraa asiakkaiden välistä kommunikaatiota ja se ei ole tyypillistä SPA-sovelluksille, jotka noudattavat ennemmin client-server kuin peer-to-peer -arkkitehtuuria. WebRTC-protokolla [33] mahdollistaa kuitenkin suoran selainten välisen kommunikaation, jolloin kaikki-kaikille -topologia olisi mahdollinen myös SPA-sovelluksille.

Hajautetut tietokannat voidaan luokitella myös homogeenisiin ja heterogeenisiin järjestelmiin. Homogeenisissä tietokannoissa jokainen hajautetun tietokannan pisteistä käyttää samaa tietokannanhallintajärjestelmää. Lisäksi pisteet ovat tietoisia toisistaan ja tekevät yhteistyötä kyselyjen suorittamisessa. Heterogeeniset tietokannat voivat käyttää eri pisteissä erilaisia tietokantajärjestelmiä ja tietomalleja. Pisteiden välinen yhteistyö ja tietoisuus toisistaan voi olla rajallisempaa kuin homogeenisissä järjestelmissä [40]. SPA-sovellukset ovat selvästi heterogeenisiä hajautettuja tietokantoja, sillä selaimella ja palvelimella käytetään erilaisia tietokannanhallintajärjestelmiä.

6 Replikointiprotokollat

Hajautetuissa tietokannoissa on käytössä useita erilaisia tapoja datan replikointiin pisteiden välillä. Kirjassa Designing data-intensive applications [38] listataan neljä erilaista keinoa replikoinnin toteuttamiseen. Vastaavat menetelmät on listattu myös PostgreSQL dokumentaatiossa aihetta käsittelevässä artikkelissa [53]. Seuraavassa esitellään nämä menetelmät ja poimitaan niistä piirteitä, jotka tekevät niistä erilaisia. Nämä ominaisuudet muodostavat mallin, jonka avulla vertaillaan web-sovelluksissa käytettyjä tiedonsiirron tekniikoita replikoituihin tietokantoihin tältä osin.

6.1 PostgreSQL replikointiprotokollat

PostgreSQL-tietokannan dokumentaatio esittelee seuraavat replikointiprotokollat: lausekeperusteinen replikointi (engl. statement-based replication), transaktiolokien lähetys (engl. write-ahead log shipping), looginen riviperusteinen replikointi (engl. logical row-based log replication) sekä laukaisinperusteinen replikointi (engl. trigger based replication). Nämä menetelmät esitellään seuraavaksi.

6.1.1 Lausekeperusteinen replikointi

Lausekeperusteisessa replikoinnissa kirjoitusoperaation vastaanottanut piste lähettää saamansa pyynnön sellaisenaan muille tietokantapisteille [38]. SQL-tietokannoissa tämä tarkoittaa siis SQL-lauseen sekä sen parametrien lähettämistä.

Vaikka tekniikka on konseptina melko yksinkertainen, se sisältää joitakin yksityiskohtia, jotka vaikeuttavat sen toteuttamista [38]. Vastaanottavan tietokantapisteen on voitava jäsentää ja suorittaa lause juuri samalla tavalla kuin lähettävä piste, jotta operaatioiden lopputulos olisi sama molemmissa pisteissä. Tämä tarkoittaa, että suoritettavat lauseet eivät saa sisältää epädeterminististä funktiokutsuja, kuten `RAND()` tai `NOW()`. Operaatiot on myöskin suoritettava täsmälleen samassa järjestyksessä jokaisessa pisteessä. Lisäksi mahdolliset sivuvaikutukset, kuten laukaisimet (engl. trigger) tai tallennetut proseduurit (engl. stored procedure) monimutkaistavat menetelmän käyttöä. Kleppmann mainitseekin, että lukuisten erikoistapausten takia usein suositaankin jotakin muuta replikointimenetelmää

[38].

PostgreSQL dokumentatiossa [53] kuvailtu malli on variaatio, jossa SQL-lauseiden välittämisen hoitaa tietokantapisteen sijasta välikerrosohjelmisto. PostgreSQL ei siis itsessään tarjoa mahdollisuutta tähän menetelmään, mutta dokumentaatio mainitsee kaksi kolmannen osapuolen ohjelmistoa, jotka tarjoavat tämän toiminnallisuuden: Pgpool-II ja Continuent tungsten.

6.1.2 Transaktiolokien lähetys

Transaktiolokien lähetys -replikointimenetelmä perustuu transaktiolokeihin eli WAL-lokeihin, jotka kirjoitetaan relaatiotietokannassa levylle transaktioiden aikana. Nämä lokit muodostavat yhdessä tilannevedosten (engl. snapshot) kanssa täydellisen varmuuskopion tietokannan datasta. Näitä lokeja voidaan käyttää myös tietokannan replikoimiseen lähettämällä lokitietueet standby-serverille. [38]

Transaktiolokit kuvaavat tietokantaan tapahtuvia muutoksia hyvin matalalla tasolla [38]. Lokissa käytetään osoittimena levylohkojen osoitteita ja muutos ilmaistaan muuttuneina bitteinä. Standby-palvelimesta muodostuu siis täsmällinen kopio master-palvelimesta jopa bittitasolla tarkasteltuna. Tekniikkaa kutsutaankin fyysiseksi replikoinniksi. Tämä tarkoittaa samalla sitä, että lokeja voidaan käyttää replikointitekniikkana vain täsmälleen samanlaisen tietokantahallintajärjestelmän välillä. Tämä vaikeuttaa esimerkiksi tietokantajärjestelmän versiopäivitysten tekemistä ilman seisokkiaikaa (engl. downtime).

6.1.3 Looginen riviperusteinen replikointi

Vaihtoehtona fyysiselle replikoinnille on olemassa looginen riviperusteinen replikointi [38]. Siinä replikointiin käytettävä formaatti ei ole niin vahvasti sidoksissa tietokantajärjestelmään kuin fyysisessä replikoinnissa. Esimerkkinä relaatiotietokannan päivitysoperaation looginen loki sisältää taulun nimen, päivitetyn rivin pääavaimen sekä päivitettyjen sarakkeiden nimet ja datan. [53].

Loogista replikointia käyttäen on helpompi ylläpitää yhteensopivuutta tietokantajärjestelmän versioiden välillä [38]. Jopa replikointi eri tietokantajärjestelmien välillä on mahdollista. Loogisia lokeja voidaan myös käyttää datan reaaliaikaiseen viemiseen toiseen järjestelmään, kuten tietovarastoon. Lisäksi looginen replikointi mahdollistaa PostgreSQL-tietokannoissa multi-master -konfiguraation käytön toisin kuin WAL-lokiin perustuva

replikointi [53].

6.1.4 Laukaisinperusteinen replikointi

Laukaisinperusteinen replikointi poikkeaa edellisestä toteutuskerrokseltaan. Kolme edellä esiteltyä replikointimenetelmää ovat tietokantajärjestelmän tarjoamia tekniikoita. Joissain tilanteissa kuitenkin voidaan tarvita räätälöidympiä ratkaisuja. Näitä voidaan toteuttaa laukaisimien ja itse kirjoitetun sovelluskoodin avulla [38]. Laukaisimien avulla voidaan suorittaa haluttuja tallennettuja proseduureja automaattisesti datan muuttuessa. Asettamalla laukaisimet vain haluttuihin tauluihin voidaan rajata replikointi vain osaan tietokannan sisällöstä. Näin voidaan esimerkiksi tallentaa tapahtunut muutos halutussa formaatissa erilliseen tietokantatauluun, josta ulkoinen ohjelma voi lukea muutokset ja lähettää ne edelleen replikoinnin kohteeseen. Tähän ulkoiseen ohjelmaan voidaan toteuttaa esimerkiksi konfliktien ratkaisuun tarvittavaa logiikkaa. Huonoina puolina tässä tekniikassa ovat järjestelmän sisäisiä replikointimenetelmiä suurempi mahdollisuus ohjelmointivirheisiin sekä suurempi määrä ylimääräistä prosessointia.

6.2 Protokollien ominaisuudet

Edellä käsiteltiin yleisesti käytössä olevia hajautettujen tietokantojen käyttämiä replikointiprotokollia. Seuraavaksi poimitaan niistä sekä replikointia käsittelevästä kirjallisuudesta ominaisuuksia, jotka ovat mielenkiintoisia protokollia vertailtaessa.

Tanenbaum jakaa [57] replikointimenetelmät kolmeen kategoriaan **propagoitavan artefaktin** mukaan. Nämä vaihtoehtoiset artefaktit ovat notifikaatio, data ja operaatio.

Päivitysoperaation toteuttanut tietokantapiste voi lähettää muille pistelle pelkän yksinkertaisen ilmoituksen eli notifikaation siitä, että tietokannan data on päivittynyt. Ilmoituksen saavan pisteen vastuulla on päivittyneen datan pyytäminen. Näin toimivat esimerkiksi invalidointiprotokollat. Lähetettävät viestit ovat hyvin pieniä, joten menetelmä voi vähentää turhaa tietoliikennettä erityisesti, jos päivitysoperaatioita tapahtuu suhteellisen paljon lukuoperaatioihin verrattuna.

Notifikaation sijaan voidaan lähettää muuttunut data sekä tarvittava määrä yksilöivää tietoa muutoksen sijoittamiseen oikeaan paikkaan. Relaatiotietokannoissa tämä tarkoittaa siis esimerkiksi taulun nimeä ja pääavainta sekä mahdollisesti sarakkeiden nimiä, kuten loogisessa replikoinnissa.

Kolmas vaihtoehto on dataan kohdistuvan operaation eli esimerkiksi SQL-lauseen lähettäminen, kuten lausekeperustaisessa replikoinnissa.

Replikointi voidaan tehdä käyttäen **push- tai pull-**mallia [57]. Push-mallissa eli työntävässä mallissa päivityksen saanut palvelin huolehtii muutoksen lähettämisestä muihin tietokantapisteisiin. Tätä käytetään etenkin järjestelmissä missä pisteet halutaan pitää mahdollisimman samansisältöisinä. Push-malli on tehokas varsinkin, jos tietokantapisteisiin tehdään paljon lukuoperaatioita suhteessa päivityksiin. Tämä toteutuu varsinkin silloin, kun yhtä tietokantapistettä käyttävät lukuoperaatiohin useat asiakkaat.

Pull-mallissa dataa käyttävä tietokantapiste on vastuussa päivitysten pyytämisestä isäntäpisteeltä. Mallin etuna on se, että isäntäpalvelimella ei tarvitse push-mallin tavoin pitää kirjaa pisteistä, joille päivitykset pitää lähettää. Pull-malli on tehokas, kun päivityksiä on paljon suhteessa lukuoperaatioihin. Tällöin jokaisen päivityksen työntäminen jokaiseen pisteeseen aiheuttaisi turhaa tietoliikennettä. Tämä pätee erityisesti silloin, kun tietokantapiste on yksittäisen asiakkaan yksityinen välimuisti.

Osa replikointimenetelmistä vaatii koko tietokannan **täydellisen replikoimisen** ja osassa voidaan valita replikoitavaksi vain osajoukko datasta. PostgreSQL-tietokannoissa looginen replikointi ja heräteperusteinen replikointi sallivat kopioimisen rajaamisen osajoukkoon tietokannan datasta. Fyysinen eli transaktiolokeihin perustuva replikointi voidaan tehdä vain koko tietokannalle.

Jokaisessa edellä esitellyssä replikointimenetelmässä on erilainen **yhteensopivuus** replikoitavien pisteiden tietokantajärjestelmien osalta. WAL-lokeihin perustuva replikointi vaatii täsmälleen samat tietokannanhallintajärjestelmät kaikissa pisteissä eli se soveltuu vain homogeenisiin hajautettuihin tietokantoihin. Muut menetelmät soveltuvat tietyin rajauksin myös heterogeeniselle järjestelmille. Lausekeperustaiseen replikointiin riittää, kun pisteet käyttävät täsmälleen samaa kyselykieltä ja kyselyt käännetään samoiksi operaatioiksi. Loogisessa replikoinnissa tietokantapisteet saattavat käyttää erilaisia kyselykieliä, mutta niillä täytyy samanlainen tietomalli tai kyky tehdä skeemojen välinen muutos osana replikointiprosessia. Herätinperusteinen replikointi on mahdollista hyvinkin erilaisten tietokantapisteiden välillä, sillä replikointilogiikka täytyy toteuttaa itse.

Lisäksi replikointitavat eroavat toisistaan **toteutuskerrokseltaan**. PostgreSQL-tietokannoissa on sisäänrakennettu tuki WAL-lokiin perustuvaan ja loogiseen replikointiin. Ne on siis toteutettu näistä menetelmistä matalimmalla tasolla eli tietokannanhallintajärjestelmässä. Herätinperusteisen replikoinnin toteuttaminen on puolestaan lähes täysin sovelluskehittäjän vastuulla. Toteutuskerrokseltaan näiden välille asettuu lauseke-

perusteinen replikointi, joka on tarjolla PostgreSQL-tietokantoihin kolmannen osapuolen välikerrosohjelmistona.

Taulukossa 6.1 on esitetty nämä piirteet tiivissä muodossa:

Piirre	**Selite**
Propagoitava asia	Replikointiprotokolla käyttävät pääosin kolmea erilaista tapaa päivitysten propagointiin: datan, operaatioiden tai notifikaatioiden lähettäminen.
Pull vai push	Palvelimelta selaimeen suuntautuva päivitysten propagointi tehdään joko selaimen aloitteesta pull-mallilla tai palvelimen aloitteesta push-mallilla
Täysi/Osittainen	PostgreSQL-tietokannan fyysinen replikointi edellyttää tietokannan kopioimista kokonaisuudessaan. Looginen replikointi mahdollistaa osittaisen replikoinnin joko rajausehdoilla (engl. filter) tai määrittelemällä halutut tietueet kyselyjen avulla.
Yhteensopivuus	Tietokantojen replikointiprotokollat voivat olla hyvin spesifejä tai yleiskäyttöisiä. Sama koskee seuraavaksi esiteltäviä webtekniikoita. Osa niistä on melko matalan tason viestiprotokollia, kun taas PouchDB on tietokanta, jolla on oma protokollansa replikoimiseen.
Toteutuskerros	Hajautettuja konfiguraatioita tukevat tietokantajärjestelmät hoitavat datan replikoinnin eikä sovelluskehittäjän tarvitse itse huolehtia siitä. Näin toimii esimerkiksi PostgreSQL WAL-lokeihin perustuvaa replikointia käytettäessä. Silloin, kun halutaan tehdä räätälöidympiä ratkaisuja esimerkiksi konfliktien ratkaisua varten, voidaan joutua kirjoittamaan replikointilogiikkaa itse.

Taulukko 6.1: Replikointiprotokollien ominaisuuksia

6.3 Vertailu SPA-sovelluksiin

Seuraavaksi vertaillaan SPA-sovellusten käyttämiä selaimen ja palvelimen välisiä kommunikaatioprotokollia edellä esiteltyihin hajautettujen tietokantojen käyttämiin replikointiprotokolliin. Protokollien ominaisuuksia on kerätty taulukkoon 6.1 ja esiteltäviä selainpuolen tekniikoita vertaillaan tämän taulukon avulla hajauttujen järjestelmien piirteisiin. Näin

saadaan muodostettua kuva tekniikoiden eroavaisuuksia ja mahdollisista ominaisuuksista, joiden kohdalla selainpuolella kannattaisi ottaa mallia tietokannoista.

Selaimen ja palvelimen välillä käytettyjen tiedonsiirtoratkaisuja on valittu tähän vertailuun seuraavin perustein. REST ja GraphQL ovat web-sovelluksissa yleisesti käytettyjä tekniikoita. XMLHttpRequest, WebSocket ja WebRTC on poimittu rajaamalla selainten tarjoamien rajapintojen [60] listauksesta tiedonsiirtoon liittyviä rajapinnat. Lisäksi joukkoon on valittu selaimessa toimiva tietokanta PouchDB, joka tarjoaa mahdollisuuden automaattiseen datan synkronoimiseen selaimen ja palvelimen välillä.

6.3.1 XMLHttpRequest ja REST-arkkitehtuuri

XMLHttpRequest (XHR) on selainten tarjoama JavaScript-rajapinta, joka mahdollistaa Ajax-pyynnöt [35]. Sen avulla selainsovellus voi ladata dataa palvelimelta päivittämättä selaimen sivua. REST [17] taas on laajasti käytetty arkkitehtuurityyli Ajax-pyyntöjen tekemiseen ja se määrittelee yleiskäyttöisen formaatin pyyntöjen tekemiseen. Ajax-pyyntöjen osoitteena käytetään halutun resurssin määrittelevää URI-tunnistetta. Haluttu operaatio ilmaistaan HTTP-standardin mukaisilla metodeilla ja tarvittava data ilmaistaan esimerkiksi JSON-muodossa.

Piirre	**XHR ja REST**
Propagoitava asia	Selain saa päivitykset palvelimelta GET-pyyntöjen vastauksina **datana**. Selaimen päivittäessä dataa se lähettää päivityksen POST-, PUT- tai PATCH-pyyntönä ja muuttunut data lähetetään pyynnön runko-osassa (engl. body) usein JSON-muodossa.
Pull vai push	REST-mallissa selain saa palvelimella tapahtuneet päivityksen vain itse pyytämällä eli kyseessä on pull-malli.
Täysi/Osittainen	HTTP-pyyntöjen avulla replikointi tehdään vain sille datalle, jonka sovelluskehittäjä määrittelee.
Yhteensopivuus	Yksinkertaisuus ja yhtenäisyys ovat olleet tärkeitä suunnitteluperiaatteista REST-mallia kehittäessä. Protokolla on hyvin yleiskäyttöinen ja laajasti yhteensopiva erilaisiin ympäristöihin.
Toteutuskerros	Replikointilogiikkan toteuttaminen on täysin sovelluskehittäjän vastuulla.

Taulukko 6.2: XHR ominaisuudet

Eheyden hallintaan liittyvänä erityispiirteenä HTTP-protokolla sisältää myös *ETag* [48] nimisen sisäänrakennetun ominaisuuden optimistisen rinnakkaisuuden hallinnan toteuttamiseksi. Siinä asiakas saa GET-pyynnön yhteydessä eräänlaisen versionumeron eli ETag-tunnisteen. GET-pyyntöä seuraavat PUT- tai PATCH-operaatiot voivat liittää tämän tunnisteen mukaan dataa muokkaavaan operaatioon. Tällöin palvelin voi varmistaa onko muokattava data ehtinyt muuttua tällä välin. Jos dataa ei ole muokattu, operaatio sallitaan.

Lisäksi HTTP-pyyntöjen etuna ovat selaimien kehittyneet ja vakiintuneet tekniikat välimuistin käyttöön. Eheyden hallinnan kannalta olennaista on mahdollisuus asettaa sallittu käyttöikä HTTP-pyynnön vastaukselle.

6.3.2 WebSocket

WebSocket-protokollassa [16] selaimen ja palvelimen välillä on jatkuva yhteys. Tämä mahdollistaa viestien lähettämisen palvelimen aloitteesta eli push-mallia käyttäen. Selainsovellus voi siis näyttää reaaliaikaista dataa ilman jatkuvaa päivitysten pyytämistä. Lähetettävien viestin formaatti on täysin vapaa ja ne voidaan lähettää teksti- tai binäärimuotoisena.

Piirre	**WebSocket**
Propagoitava asia	WebSocket ei määrittele viestin formaattia millään tavalla. Viestit voivat olla siis **dataa** tai **operaatioita**. Lisäksi myös **notifikaatiot** voivat olla käytännöllisiä, sillä palvelin voi lähettää viestejä selaimeen.
Pull vai push	WebSocket-protokollan olennainen piirre on push-ominaisuus eli mahdollisuus päivitysten propagoimiseen palvelimelta käsin toisin kuin HTTP-pyynnöissä, jotka ovat aina asiakkaan aloittamia.
Täysi/Osittainen	WebSocket-protokollan avulla replikointi tehdään vain sille datalle, jonka sovelluskehittäjä määrittelee.
Yhteensopivuus	HTTP-pyyntöjen tapaan WebSocket on hyvin **yleiskäyttöinen** eikä sitä ole erityisesti suunniteltu käytettäväksi jonkin tietyn tietokannan kanssa. WebSocket-protokolla ei ota kantaa lähetettävien pyyntöjen sisältöön, joten ilman REST-mallin kaltaista arkkitehtuurimallia WebSocket-protokollalla tehtävä replikointi voi olla hyvinkin sovellusspesifiä protokollan yleiskäyttöisyydestä huolimatta.
Toteutuskerros	Replikointilogiikkan toteuttaminen on täysin sovelluskehittäjän vastuulla.

Taulukko 6.3: WebSocket

6.3.3 GraphQL

GraphQL [21] on korkeamman tason protokolla kuin XHR tai WebSocket. Sitä verrataan usein REST-malliin, joka on abstraktiotasoltaan parempi vertailukohta. GraphQL määrittelee kyselykielen ja formaatin, jossa pyyntöjen vastaukset annetaan. REST-mallin tavoin kommunikaatioprotokollana käytetään usein HTTP-pyyntöjä vaikka mallia ei ole siihen sidottu.

Piirre	**GraphQL**
Propagoitava asia	Selaimesta palvelimen suuntaan tapahtuvat päivitykset propagoidaan GraphQL-kielisinä **operaatioina**, jotka esitetään JSON-formaatissa. Datan päivittämiseen GraphQL-kieli ei kuitenkaan tarjoa kovin monipuolisia operaatioita, vaan kyselyt muistuttavat etäkäsittelykutsuja (engl. remote procedure call), jolle annetaan operaation nimi ja parametrit. Palvelimelta selaimeen suuntaan päivitykset propagoituvat **datana** REST-mallin tapaan.
Pull vai push	Yleensä GraphQL-operaatiot ovat **pull**-tyyppisiä, mutta myös **push**-operaatiot ovat mahdollisia *subscription*-nimisen operaation avulla. Normaalisti GraphQL-operaatio tehdään HTTP-pyyntöjen avulla, mutta push-operaatiot tehdään WebSocket-protokollaa käyttäen.
Täysi/Osittainen	GraphQL:n avulla replikointi tehdään vain sille datalle, jonka sovelluskehittäjä määrittelee.
Yhteensopivuus	GraphQL sopii käytettäväksi useiden erilaisten tietokanta- ja tilanhallintajärjestelmien kanssa. Jos käytetään selaimessa tehtävään tilanhallintaan GraphQL Apollo -kirjastoa, voidaan paikalliseen tilaan kohdistuvat operaatiot tehdä samalla kielellä kuin palvelimelle tehtävät pyynnöt.
Toteutuskerros	Replikoinnin toteuttaa sovelluskehittäjä ja mahdollisesti osittain tilanhallintajärjestelmä. Jos käytetään tilanhallintaa Redux-kirjastoa ja palvelintietokantana PostgreSQL-tietokantaa, on replikoinnin suorittaminen GraphQL-operaatioina täysin sovelluskehittäjän vastuulla. Käyttämällä Apollo-kirjastoa voidaan ajatella, että replikointi on osittain käytettyjen kirjastojen hoitamaa. Sovelluskehittäjä voi kirjoittaa pyyntöjä välittämättä vastaanko pyyntöön paikallisesta datasta vai haetaanko data palvelimelta.

Taulukko 6.4: GraphQL ominaisuudet

Seuraavasta esimerkistä nähdään GraphQL-kielisten päivitysoperaatioiden eli mutaatioiden rakenne:

```
const ADD_COMMENT = gql‘
  mutation addComment($itemId: String!, $author: String!, $comment: String!) {
    addNewComment(
      itemId: $itemId,
      author: $author,
      comment: $comment
    ) {
      commentId
    }
  }
‘
```

Esimerkissä alustetaan mutaatio `addComment` gql-notaatiolla muuttujaan `ADD_COMMENT`. Mutaatiolause sisältää operaation palvelimella käytettävän nimen `addNewComment` sekä tarvittavat parametrit `itemId`, `author`, `comment`. Lisäksi lauseessa määritellään mitä tietoja halutaan operaation palautusarvona. Tässä tapauksessa määritellään palautettavaksi uuden kommentin saama tunniste `commentId`. Selaimen tekemä mutaatiopyyntö ei siis sisällä HTTP-pyyntöjen metodien kaltaista semantiikkaa siitä, että onko operaatiossa kyse uuden lisäämisestä, vanhan muokkaamisesta vai tiedon poistamisesta.

6.3.4 PouchDB

Edellisistä poiketen PouchDB [54] ei ole vain protokolla selaimen ja palvelimen väliseen tiedonvaihtoon vaan kokonainen tietokantajärjestelmä. Se on JavaScript-implementaatio CouchDB-tietokannasta. CouchDB on dokumenttitietokanta ja sen erityispiirteenä on mahdollisuus multi-master -replikointiin. Replikointi tehdään HTTP-protokollaa käyttäen, joten selaimessa toimiva PouchDB voidaan suoraviivaisesti synkronoida palvelimella olevan CouchDB-tietokannan kanssa.

Piirre	**PouchDB**
Propagoitava asia	Palvelimen CouchDB-tietokanta lähettää asiakkailleen **notifikaation** dataan tulleista muutoksista. Asiakas pyytää halutessaan päivitykset erillisenä pyyntönä.
Pull vai push	PouchDB mahdollistaa päivitysten propagoimisen palvelimelta selaimeen push-mallilla. Tämä toteutetaan oletuksena HTTP-pyynnöillä *long polling* -tekniikalla, mutta myös WebSocket-implementaatio on olemassa. [58]
Täysi/Osittainen	Muista listatuista protokollista poiketen PouchDB replikoi oletuksena kaiken tietokannan datan. Replikoitavaa dataa voidaan kuitenkin rajata haluttuun osajoukkoon.
Yhteensopivuus	PouchDB käyttää CouchDB:n replikointirajapintaa ja protokollaa. Samaa protokollaa voisi olla mahdollista käyttää muidenkin tietokantojen välillä sopivien välikerrosohjelmistojen avulla, mutta protokolla ei ole samalla tavalla yleiskäyttöiseksi suunniteltu kuin REST tai GraphQL
Toteutuskerros	Myös toteutuskerroksen osalta PouchDB on poikkeava tässä joukossa. Replikointi on täysin järjestelmän toteuttamaa eikä sovelluskehittäjän tarvitse itse määritellä replikointiin vaadittavia operaatioita.

Taulukko 6.5: PouchDB ominaisuudet

6.3.5 WebRTC

Lopuksi mainittakoon vielä WebRTC-protokolla [33]. Se on teknologia, jonka avulla selainten välille voidaan luoda vertaisyhteys (engl. peer-to-peer connection). Se on kehitetty erityisesti ääntä ja kuvaa eli esimerkiksi videopuheluja varten, mutta sitä voidaan käyttää myös muunlaisen datan lähettämiseen ja vastaanottamiseen.

Piirre	**WebRTC**
Propagoitava asia	WebRTC ei määrittele viestin formaattia millään tavalla.
Pull vai push	WebRTC:n jatkuva yhteys mallistaa push-mallin käytön
Täysi/Osittainen	Myös WebRTC:n avulla replikointi tehdään vain sille datalle, jonka sovelluskehittäjä määrittelee.
Yhteensopivuus	WebRTC on suhteellisen tuore teknologia eikä se ole vielä saavuttanut W3C Recommendation -tasoa. Se on kuitenkin laajasti selainten tukema. Matalan tason protokollana WebRTC ei ole sidottu mihinkään tiettyyn tilanhallintajärjestelmään.
Toteutuskerros	Replikointilogiikkan toteuttaminen on täysin sovelluskehittäjän vastuulla.

Taulukko 6.6: WebRTC ominaisuudet

6.4 Havaintoja

Esitellyissä menetelmissä käytettiin replikointiin sekä datan että notifikaatioiden lähettämistä. Lisäksi GraphQL:n voidaan katsoa lähettävän operaatioita, mutta ne muistuttavat kenties enemmän funktiokutsuja kuin varsinaisia datan päivittämiseen käytettyjä lauseita. Aliluvussa 6.1.1 esitellyssä lausekeperusteisessa replikoissa käytetään samoja SQL-lauseita datan päivittämiseen ja päivitysten propagointiin. Tämä voisi olla mielenkiintoinen lähetysmistapa myös SPA-sovelluksiin.

Hajautetuissa tietokannoissa päivitykset propagoidaan yleensä push-mallilla. SPA-sovelluksissa pull-malli on yleinen, mutta push-malli on myös mahdollinen. Tietokannoissa käytettyjen ratkaisujen valossa näyttäisi siltä, että myös SPA-sovelluksissa kannattaisi pyrkiä käyttämään push-mallia.

Käsitellyissä tekniikoissa oli mukana sekä laajasti yhteensopivia yleiskäyttöisiä tekniikoita, kuten XHR-pyynnöt ja REST-arkkitehtuuri sekä spesifejä ratkaisuja, kuten PouchDB. Sa-

moin hajauttujen tietokantojen piirissä on myös molempia lähestymistapoja käytössä. Jos verrataan omilla alueilla erittäin suosittuja REST-rajapintoja ja PostgreSQL-tietokantoja niin saadaan jonkinlaista eroavaisuutta esille. Selainpuolen replikointityyppinen tiedonsiirto hoidetaan usein yleiskäyttöisellä, mutta runsaasti ohjelmointia vaativalla HTTP-protokollalla ja palvelinpuolella PostgreSQL käyttää oletuksena ei-yleiskäyttöistä, mutta automaattista WAL-lokeihin perustuvaa menetelmää. Tätä voinee käyttää argumenttina ei-yleiskäyttöisten menetelmien käyttämiseen jossain tilanteissa myös selainpuolella.

Selkeä eroavaisuus selain- ja palvelinpuolen välillä on replikonnin toteutuskerroksessa. Useimmissa käsitellyistä selainpuolen tekniikoissa varsinainen replikointilogiikan toteuttaminen jää täysin sovelluskehittäjän vastuulle. Palvelinpuolella tilanne on päinvastainen ja käytetyimmät replikointimenetelmät ovat tietokantajärjestelmän avulla automatisoituja. PouchDB:ssä synkronointi on automaattista, mutta se ei kuitenkaan ole kovin laajasti käytössä. Tämän on sanottu johtuvan muun muassa ongelmista konfliktien ratkaisussa ja kyselyissä käytettävästä monille vieraasta MapReduce-mallista [39]. PouchDB:n filosofia on kuitenkin saanut kiitosta [39] ja automaattinen replikointi on todettu toimivaksi palvelinpuolen tietokannoissa. Näillä perustein PouchDB:n kaltainen automaattisesti replikoiva tietokanta voisi olla tietyin muutoksin todella käyttökelpoinen SPA-sovellusten tilanhallintaan.

Lisäksi selainten menetelmät replikointiin eroavat tietokantojen vastaavista siten, että ne eivät mahdollista replikoinnin keskeyttämistä ja jatkamista, kuten transaktiolokien lähetys -menetelmässä. Hajautetun tietokannan pisteiden välisen verkkoyhteyden katketessa transaktiolokien lähetys joudutaan keskeyttämään kunnes yhteys palaa. Tämä ei kuitenkaan estä uusien transaktioiden suorittamista, mikäli käytetään asynkronista replikointia. Verkkokatkoksen aikana transaktiolokit tallennetaan normaaliin tapaan ja niiden lähettämistä muihin pisteisiin jatketaan verkkoyhteyden palattua. Myös SPA-sovelluksissa vastaava olisi mahdollista esimerkiksi tallentamalla jonkinlaista transaktiolokia paikalliseen IndexedDB-tietokantaan ja toteuttamalla replikoinnin näitä lokeja lähettämällä.

7 Eheyden hallinta

Eheyden hallinta on yksi tietokantojen olennaisista ominaisuuksista ja aihetta on tutkittu ja kehitetty jo vuosikymmeniä. Se koskee niin keskitettyjä kuin hajautettuja järjestelmiä. Monimutkaistuvat web-sovellukset sisältävät jatkuvasti enemmän dataa selaimessa ja sen hallitseminen muistuttaa tietokannan operoimista. WebSQL ja IndexedDB ovat kirjaimellisesti selaimessa toimivia tietokantoja.

Tässä luvussa käydään läpi eheyden hallintaan kehitettyjä menetelmiä sekä malleja ja arvioidaan niitä SPA-sovellusten kontekstissa. Tavoitteena on rakentaa ymmärrystä siitä, mitkä näistä aiheista tulisi huomioida SPA-sovelluksia kehitettäessä. Näin pyritään löytämään seikkoja, joissa SPA-sovelluksiin voisi ottaa mallia tietokantojen ominaisuuksista. Tarkastelussa oletetaan lähtökohtaisesti SPA-sovellus, joissa käyttöliittymä on toteutettu React-kirjastolla, sovelluksen paikallisen tilan hallintaan käytetään Redux-kirjastoa, kommunikaatio palvelimelle tapahtuu REST-rajapinnan kautta ja palvelinpuolella on yksinkertainen web-palvelin sekä relaatiotietokanta.

SPA-sovelluksia arvioidaan neljästä tietokantojen eheyden hallintaan liittyvästä näkökulmasta: ACID-ominaisuudet, eristyvyysanomaliat, hajautetun eheyden mallit sekä konfliktien ratkaiseminen. Akronyymi ACID on tietokantatransaktioita [22] määrittelevä kulmakivi. Se määrittelee transaktioille neljä ominaisuutta, joiden avulla tietokannan data säilytetään eheänä. Eristyvyysanomaliat ovat ilmiöitä, jotka voivat ilmetä transaktioiden rinnakkaisessa suorituksessa. Niiden hallitsemiseksi SQL-standardissa on eri vahvuisia eristyvyystasoja. Myös replikoitujen tietokantojen pisteiden väliselle eheydelle on kehitetty eri vahvuisia eheysmalleja. Tiukat eristyvyystasot ja eheysmallit pitävät datan varmemmin eheänä, mutta ne vaativat enemmän koordinaatiota. Hajautetussa tietokannassa, jossa on käytössä asynkroniset kirjoitusoperaatiot, voidaan törmätä tilanteeseen, jossa samaan aikaan eri pisteissä tehdyt päivitykset ovat ristiriidassa keskenään. Näissä tilanteissa tarvitaan konfliktien ratkaisumenetelmiä.

7.1 ACID-ominaisuudet

ACID-vaatimusten [24] mukaan tietokantatransaktioiden tulee olla atomisia (engl. atomic), oikeellisia (engl. consistent), pysyviä (engl. durable) ja eristettyjä (engl. isolated).

ACID-vaatimukset koskevat niin yksittäisen palvelimen tietokantaa kuin hajautettuja tietokantoja. SPA-sovellusten kohdalla voidaan siis tarkastella miten vaatimukset toteutuvat paikallisen tilan ja palvelimelle tapahtuvien pyyntöjen kohdalla.

7.1.1 Atomisuus

Atomisuusvaatimus tarkoittaa, että transaktio täytyy suorittaa kokonaisuudessaan tai ei ollenkaan. Keskeneräiseksi jäävä operaatio jättäisi tietokannan epäeheään tilaan.

Redux-storeen tehtävät tilan muutokset ovat aina atomisia, sillä muutokset toteutetaan immutable-tietorakenteeseen eli päivitettävää osaa sovelluksen tilasta ei muokata suoraan vaan siitä tehdään uusi versio [6]. Kun tarvittavat päivitykset on tehty, muokkaukset tehnyt reducer-funktio palauttaa uuden version tilasta ja se asetaan aktiiviseksi.

XHR-pyyntöjen ja REST-rajapinnan kohdalla atomisuus on monimutkaisempi vaatimus. Yksittäisen, yhteen tietoalkioon kohdistuvan XHR-pyynnön kohdalla atomisuus toteutuu helposti, mikäli palvelin tekee saapuvasta HTTP-pyynnöstä yhden atomisen tietokanta-transaktion. Atomisuuden säilyttäminen vaatii enemmän tarkkuutta, jos operaatio kohdistuu useisiin tietueisiin. Seuraava esimerkki havainnollistaa asiaa.

Yksittäistä tietuetta voidaan muokata HTTP PATCH -operaation avulla. Esimerkissä muokataan huoneen 105 tietoja. Huoneen `description`-kenttään päivitetään teksti ”Huone on remontissa”. Päivitettävä tietue määritellään pyynnön URL-osoitteessa ja uudet tiedot liitetään pyynnön runko-osaan. Operaation HTTP-pyyntö näyttää tältä:

```
PATCH https://example.com/api/room/105/
{ description: "Huone on remontissa" }
```

Useampaa tietuetta on mahdollista päivittää tekemällä kullekin oma edellisen kaltainen HTTP-pyyntö, mutta erillisten pyyntöjen atomista toteutumista ei voida taata. Tämän sijaan operaatiot tulisi paketoida yhteen HTTP-pyyntöön. REST-mallissa käsiteltävä resurssi määritellään pyynnön osoitteessa eikä mallissa ole selkeää tapaa määritellä operaatiolle useita kohteita. Tällöin sovelluskehittäjän on helppo päätyä tekemään useita HTTP-pyyntöjä tai kehittämään oma syntaksi useiden tietuiden päivittämiseen. Tähän tarkoitukseen on kuitenkin kehitetty myös erillinen standardi. RFC 6902 [52] määrittelee JSON PATCH notaation, joka soveltuu useiden objektien käsittelyyn. Seuraava esimerkki muokkaa huoneiden 105 ja 106 kuvauksia.

```
PATCH https://example.com/api/room/
[{
    op: "replace",
    path: "/105/description/",
    value: "Huoneet 105 ja 106 ovat remontissa"
 }, {
    op: "replace",
    path: "/106/description/",
    value: "Huoneet 105 ja 106 ovat remontissa"
}]
```

7.1.2 Oikeellisuus

Relaatiotietokannoissa sovelluskehittäjä laatii datalle skeeman ja eheysrajoitteet (engl. integrity constraint), joiden avulla data säilytetään halutussa muodossa. Tietokantajärjestelmä varmistaa, että data tallennetaan skeeman mukaisesti ja että tietokantaan luodut eheysrajoitteet pysyvät voimassa transaktion sitoutuessa. Tämä koskee muun muassa sarakkeiden tietotyyppejä, vieras- ja pääavainrajoitteita sekä uniikkius- ja arvojoukkorajoitteita. Oikeellisuusominaisuus toteutuu, kun transaktio muuttaa tietokannan tilaa eheästä tilasta toiseen eheään tilaan.

Redux-kirjaston kohdalla operaatioiden oikeellisuuden varmistaminen on pääosin sovelluskehittäjän vastuulla. Redux ei itsessään tarjoa mahdollisuutta relaatiotietokantojen kaltaisten eheysrajoitteiden asettamiseen. Normalizr-kirjaston avulla on mahdollista luoda Redux-storen datalle skeema [3] ja myös TypeScript-ohjelmointikielen avulla on mahdollista luoda tyypitykset sovelluksen käyttämälle datalle [4]. Muita kuin skeemaan liittyviä eheysrajoitteita ei vaikuta juuri olevan yleisesti käytössä Redux-kirjaston yhteydessä. Aiheeseen liittyviä kirjastoja ei ole listattu ainakaan Redux-dokumentaation Ecosystem-osiossa eikä niitä löydy npm-palvelusta hakusanoilla “redux integrity constraints” tai “redux consistency”. Tässä kohtaa SPA-sovelluksiin voisi siis ottaa mahdollisesti mallia relaatiotietokannoista. Erityisesti offline-tilassa tehtäviä päivityksiä mahdollistavat sovellukset voisivat hyötyä eheyden varmistamisesta jo selaimessa.

7.1.3 Pysyvyys

Jotta transaktion vaikutus olisi pysyvä myös sovelluksen kaatuessa tai muussa ongelmatilanteessa, on tieto transaktiosta persistoitava transaktion aikana. Relaatiotietokannoissa

tämä toteutetaan transaktiolokien avulla.

Kaaviossa 3.4 esitetyn kaltaisessa yksinkertaisessa SPA-sovelluksen arkkitehtuurissa kaikki persistointi tapahtuu tietokantapalvelimen avulla. Operaatiot lähetetään HTTP-pyyntönä sovelluspalvelimelle, joka huolehtii niiden tallentamisesta tietokantaan.

Normaalisti Redux-storen tila on tallessa vain selaimen muistissa, joten vain siihen paikallisesti tehtävät operaatiot eivät ole pysyviä. SPA-sovelluksissa operaatioita on kuitenkin mahdollista persistoida myös paikallisesti käyttäjän laitteelle selainten tarjoamien rajapintojen, kuten localStoragen ja IndexedDB:n, avulla. Reduxin kanssa näitä rajapintoja voidaan käyttää erilaisten kirjastojen avulla. Redux dokumentaation Ecosystem-sivulla [5] on listattu kirjastot Redux-persist, Redux-storage ja Redux-offline. Ne mahdollistavat useiden paikallisten tallennusmekanismien (engl. storage engine) käytön Reduxin yhteydessä.

Redux-tilaa paikallisesti persistoivien kirjastojen ja transaktiolokien käytön välillä on kuitenkin selkeä ero. Reduxin sisältämä tila persistoidaan kokonaisuudessaan jokaisen operaation yhteydessä tai ajoittain, mutta relaatiotietokannat tallentavat ainoastaan minimaalisen lokitietueen transaktion aikana. Transaktiolokien tallentaminen on tehokasta, mutta tietokannan kaatuessa lokimenetelmä vaatii tietokannan tilan rakentamisen uudelleen operaatio kerrallaan. Redux-kirjastojen tapa tallentaa koko tilaobjekti on yksinkertainen, mutta se voi muodostua pullonkaulaksi.

7.1.4 Eristyvyys

Eristyvyys tarkoittaa sitä, että transaktioiden rinnakkainen suorittaminen ei vaikuta niiden lopputulokseen. Tuloksen on siis oltava sama kuin, jos transaktiot olisi suoritettu peräkkäisjärjestyksessä. Relaatiotietokannoissa eristyvyysanomaliat ovat mahdollisia mikäli transaktiot muokkaavat dataa paikallaan eivätkä käytä riittäviä lukitusmenetelmiä. Tällöin rinnakkaiset operaatiot pääsevät vaikuttamaan toistensa suoritukseen.

Web-selaimessa tapahtuvat operaatiot ovat luonnostaan toisistaan eristyneitä, sillä JavaScriptia suoritetaan yksisäikeisesti [18]. Kaikki Redux-storen tilaan tapahtuvat muutokset tapahtuvat tarkassa järjestyksessä yksi kerrallaan [6].

Optimistisia päivityksiä käytettäessä on kuitenkin mahdollisuus esimerkiksi likaiseksi lukemiseksi kutsuttuun anomaliaan. Tästä ja muista eristyvyysanomalioista tarkemmin seuraavaksi.

7.2 SQL:n eristyvyystasot ja anomaliat

ACID-akronyymin vaatimus transaktioiden täydellisestä eristyvyydestä on raskas ja sitä keventämään on kehitetty löyhempiä eristyvyystasoja. SQL-standardissa [31, 10] määritellyt transaktioiden eristyvyystasot mahdollistavat asteittaisen eristyvyyden relaksoinnin niin, että vain hallittu joukko eristyvyysanomalioita on mahdollisia. Standardissa määritellään kolme anomaliaa: likainen lukeminen, toistokelvoton luku sekä haamuilmiö. Näiden lisäksi kirjallisuudessa mainitaan usein anomalia nimeltä kadonnut päivitys [38]. Tässä luvussa esitellään nämä anomaliat ja arvioidaan ovatko ne mahdollisia SPA-sovellusten kontekstissa.

7.2.1 Likainen lukeminen

Tämä anomalia ilmenee, kun kaksi yhtäaikaista operaatiota käsittelee samaa dataa. Transaktio A muokkaa dataa, mutta ei vielä sitouta muutosta commit-käskyllä. Ilman rinnakkaisuuden hallintaa toinen transaktio B voi lukea tämän sitouttamattoman muutoksen. Jos transaktio A peruuntuu, niin transaktio B on lukenut virheellistä dataa.

Reduxin yhteydessä vastaavaa ei pääse tapahtumaan, sillä operaatioita ei erikseen sitouteta eikä keskeneräisiä muutoksia ole luettavissa immutable-tietorakenteen ansiosta. SPA-malli mahdollistaa kuitenkin myös optimistiset päivitykset (engl. optimistic update) paikalliseen tilaan. Tämä tarkoittaa Redux-storen tilan päivittämistä jo ennen kuin siihen liittyvään HTTP-pyyntöön on vastattu. Tällöin seuraavanlainen skenario johtaisi likaisen lukemisen anomaliaan:

```
- Operaatio A tekee optimistisen päivityksen paikalliseen dataan
  ja lähettää pyynnön palvelimelle.
- Operaatio B lukee tämän optimistisesti tehdyn päivityksen osana laajempaa
  yhteenvetokyselyä. Tämän kyselyn tulos tallennetaan palvelimelle.
- Palvelin hylkää operaation A pyynnön ja optimistinen päivitys peruutetaan.
- Operaatio B on tallentanut virheellisen raportin.
```

Tämän esimerkin tilanne voidaan välttää tekemällä yhteenvetokyselyt palvelimella ja sellaisen tekeminen selaimessa onkin kenties epätavallista. Esimerkki kuitenkin näyttää, että optimististen päivitysten yhteydessä on oltava tarkkaavainen paikallisen datan kanssa, sillä se voi sisältää sitoutumatonta dataa.

7.2.2 Toistokelvoton luku ja haamuilmiö

Toistokelvoton luku koskee etenkin operaatioita, jotka lukevat paljon dataa ja ovat näin pitkäkestoisia. Tällöin toinen operaatio voi ehtiä tekemään päivityksen luettuun dataan, ennen kuin ensimmäinen operaatio on päättynyt. Jos ensimmäinen operaatio toistettaisiin juuri ennen sen päättymistä, lopputulos olisi erilainen.

Reduxin kontekstissa tämä anomalia ei pääse esiintymään, sillä selain suorittaa JavaScript-koodia yksisäikeisesti ja synkronisesti, joten operaatiot eivät tapahdu rinnakkain. Palvelimella tietokanta huolehtii toistokelvottomien lukujen estämisestä tarjoamalla riittävän eristyvyystason kaikille transaktioille.

Anomalia koskee kuitenkin jossain määrin myös SPA-sovelluksia silloin, kun haetaan dataa palvelimelta. Toistokelvottoman lukemisen kaltainen anomalia on mahdollinen mikäli selaimen paikallinen tila koostetaan useiden HTTP-pyyntöjen ja tietokantatransaktioiden avulla. Tällöin koostettu paikallinen tila ei välttämättä kuvaa mitään eheää tietokannan tilannevedosta (engl. snapshot), kun pyyntöjen välissä tietokannan data voi päivittyä muiden käyttäjien operaatioiden johdosta. Mikäli selaimeen halutaan eheä tilannevedos, on käytettävät REST-rajapinnoille tyypillisten tietuekohtaisten kyselyjen sijaan räätälöidympiä ratkaisuja.

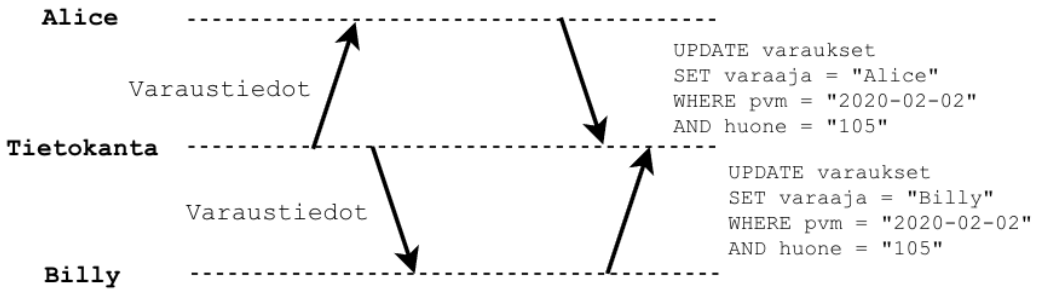
Haamuilmiö on erikoistapaus toistokelvottomasta lukemisesta, jossa toinen transaktio lisää tai poistaa rivejä niiden päivittämisen sijaan. Toistokelvottomasta lukemista esitetyt SPA-kontekstia koskevat havainnot pätevät myös haamuilmiön kohdalla.

7.2.3 Kadonnut päivitys

Tämä anomalia eroaa edellisistä merkittävästi, sillä kyseessä on kirjoittamiseen liittyvä anomalia siinä missä edelliset liittyivät lukuoperaatioita tekevien transaktioiden kohtaamiin anomaliohin. Kadonnut päivytys on mahdollinen, kun kaksi yhtäaikaista transaktiota päivittää samaa dataa ja asettaa jonkin tietyn arvon samalle tietueelle.

Rinnakkaisuuden puuttuminen estää tämänkin anomalian ilmenemisen Redux-kontekstissa paikallisissa operaatioissa. Kadonneen päivityksen anomalia on kuitenkin mahdollinen useiden käyttäjien käsitellessä samaa dataa lähes samaan aikaan. Päivitysoperaatiot, jotka asettavat tietueelle tietyn arvon, saattavat erehdyksessä ylikirjoittaa muiden tekemiä päivityksiä mikäli riittäviä varokeinoja ei käytetä. Kuvassa 7.1 kaksi käyttäjää tarkastelee huonevarauksia samanaikaisesti. Molemmat näkevät huoneen

105 vapaana haluamanaan päivänä ja tekevät siihen varauksen. Käyttäjän 2 tekemä varaus jää voimaan, sillä järjestelmä ei tässä esimerkissä varmista, että onko varausta jo olemassa. Käyttäjän 1 tekemä varaus on näin kadonnut päivitys.



Kuva 7.1: Kadonnut päivitys -anomalia

Vastaava anomalia on mahdollinen tietokantasovellukissa yleensäkin. SPA-sovelluksissa voidaan hyödyntää HTTP-protokollaan kuuluvia ETag-tunnisteita tämän anomalian välttämisessä. Niiden avulla palvelin tunnistaa selaimelta saapuvasta päivitysoperaatiosta, että onko selaimella ollut tuorein versio datasta, jota se on päivittämässä. Jos näin ei ole, pyyntö voidaan hylätä ja estää mahdollinen ylikirjoitus.

7.3 Hajautettujen tietokantojen eheysmallit

Hajautetuissa tietokannoissa eristyvyyden ja atomisuuden turvaaminen vaatii vielä enemmän koordinaatiota kuin keskitetyissä yhden pisteen tietokannoissa. Runsas koordinointi kuitenkin heikentää tietokantajärjestelmän suorituskykyä ja saatavuutta. Aihetta on tutkittu jo vuosikymmeniä [41] ja sitä käsitellään lukuisissa hajautettujen järjestelmien oppikirjoissa [20, 40, 57]. Erityisesti CAP-teoreema teki tämän ongelman tunnetuksi.

Valinta eheyden ja saatavuuden välillä ei kuitenkaan ole mustavalkoinen vaan on olemassa joukko eri tasoisia eheysmalleja, joita voidaan soveltaa hajautetuissa tietokannoissa [36]. Nämät mallit muistuttavat jossain määrin edellä käsiteltyjä eristyvyystasoja, mutta ne eivät suoranaisesti liity toisiinsa. Eristyvyystasot koskevat transaktioiden rinnakkaista suorittamista. Hajautettujen tietokantojen eheysmallit liittyvät tietokannan pisteiden välisen yhdenmukaisuuden hallintaan tietoliikenteen viiveistä ja häiriöistä huolimatta. Seuraavaksi esitellään kirjaan Distributed Systems: An Algorithmic Approach [20] valitut viisi erilaista eheysmallia ja arvioidaan niiden soveltuvuutta SPA-sovellusten kontekstiin.

Mallit muodostavat jatkuvan skaalan siten, että aluksi esitellään hyvin tiukka malli ja sen vaatimuksia helpotetaan asteittain. Näin saadaan erilaisia malleja, joista voidaan valita sovelluksen tarpeiden mukaan riittävä taso.

7.3.1 Ehdoton eheys

Tämän joukon tiukin eheysmalli on nimeltään ehdoton eheys [20] (engl. strict consistency). Tässä mallissa kaikkien päivitysten tulee olla kaikkien prosessien luettavissa välittömästi reaaliajassa. Hajautetuissa ympäristöissä tämä ei ole kuitenkaan saavutettavissa viestinvälityksen aiheuttaman viiveen takia. Tämä toimii hajautettujen järjestelmien kohdalla enemmänkin lähtökohtana, josta lähdetään väljentämään ehtoja, kunnes löydetään sopivan tasoinen malli.

7.3.2 Linearisoituvuus

Linearisoituvuudessa [25] (engl. linearizability) ei vaadita operaatioiden välitöntä näkyvyyttä jokaiselle prosessille. Jokaisen prosessin on kuitenkin havaittava operaatiot samassa järjestyksessä. Dataan tehdyt luku- ja kirjoitusoperaatiot muodostavat siis jäljen (engl. trace), joka määrittelee järjestyksen, jossa operaatiot ovat tapahtuneet. Tämän yhden kokonaisjärjestyksen (engl. single total order) tulee esittää kunkin prosessien suorittamat operaatiot niiden alkuperäisessä todellisessa järjestyksessä. Operaatiolla on siis oltava luotettavat aikaleimat ja operaatioiden tulee olla aikaleimojen mukaisessa järjestyksesä muodostettavassa kokonaisjärjestyksessä.

SPA-sovelluksissa voidaan muodostaa operaatioista keskitetyn tietokannan avulla kokonaisjärjestys, joka huomioi myös reaaliajan. Reaaliaikana joudutaan tosin käyttämään luultavasti selainsovelluksen luomien aikaleimojen sijaan pyyntöjen saapumisaikoja palvelimelle, jolloin malli muistuttaa kenties enemmän peräkkäiseheyttä. Selaimelta saapuvien pyyntöjen aikaleimoihin ei välttämättä voi luottaa, sillä laitteiden kellot eivät välttämättä ole samassa ajasta. Lisäksi aikaleimojen väärentäminen olisi melko helppoa. Linearisoituvuuden toteuttaminen SPA-sovelluksissa vaatisi siis jonkinlaisen ratkaisun aikaleimojen luotettavuuden varmistamiseen.

7.3.3 Peräkkäiseheys

Kun linearisoituvuudesta jätetään pois vaatimus aikaleimojen mukaisesta järjestyksestä eri prosessien operaatioille saadaan peräkkäiseheys [25] (engl. sequential consistency). Yksittäisten prosessien tekemien operaatioiden järjestystä on edelleen noudatettava ja operaatiosta on muodostettava yksi kokonaisjärjestys, jonka mukaisesti kaikkien prosessien tulee havaita kirjoitusoperaatiot. Tämä kokonaisjärjestys ei siis välttämättä ole sama kuin operaatioiden todellinen järjestys, mutta kaikki osapuolet näkevät operaatioiden tapahtuvan tässä järjestyksessä.

SPA-sovelluksissa tietokanta muodostaa jaetun kokonaisjärjestyksen. Jos kukin prosessi eli selainsovellus lähettää omat päivitysoperaationsa järjestyksessä ja yksi kerrallaan, tietokantaan muodostuu peräkkäiseheyden mukainen järjestys operaatioista. Peräkkäiseheys ei täyty, jos selain lähettää useita HTTP-pyyntöja palvelimelle rinnakkain ja pyynnöt suoritetaan palvelimelle eri järjestyksessä kuin oli tarkoitettu.

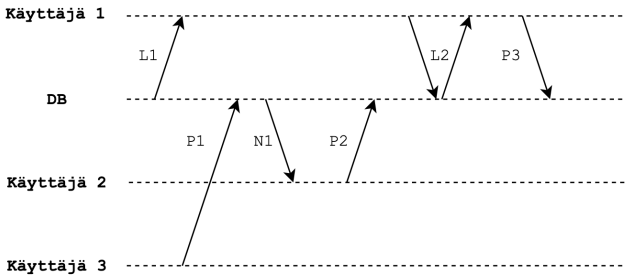
Vaikka tietokantaan saadaan muodostettua jaettu kokonaisjärjestys voidaan SPA-sovelluksissa kuitenkin rikkoa peräkkäiseheyden mallia melko helposti. Oleellinen osa on vaatimus kirjoitusoperaatioiden havaitsemisesta oikeassa järjestyksessä. Kuvitellaan, että käyttäjä Alice avaa SPA-sovelluksen klo 14.00 ja selaimeen on ladataan sovelluksen tila tietokannasta. Alice selailee sovellusta hetken ja tekee klo 14:05 jonkin dataa kirjoittavan operaation. Operaatio lähetetään palvelimelle, joka tallentaa tiedon tietokantaan ja vastaa operaation onnistuneen. Nyt selain toteuttaa päivityksen myös paikalliseen tilaansa. Nyt muiden käyttäjien klo 14.00-14.05 välillä tekemiä päivityksiä ei ladattu Alicen selaimeen. Alice näkee siis oman päivityksensä ennen tuolla välillä tehtyjä päivityksiä, vaikka ne ovat palvelimelle ja tulisi siten näyttää ennen Alicen klo 14:05 tekemää päivitystä.

7.3.4 Kausaalinen eheys

Kausaalisessa eheydessä [7] (engl. causal consistency) toisistaan riippuvat kirjoitusoperaatiot tulee esittää oikeassa järjestyksessä. Operaation katsotaan riippuvan toisesta mikäli jälkimmäinen on nähnyt ensimmäisen operaation tuloksen. Näin yhden prosessin operaatiot ovat toisistaan riippuvia eli edellisten mallien tapaan yhden prosessin operaatiot on esitettävä todellisessa järjestyksessä. Peräkkäisjärjestyksestä poiketen kausaalisen eheyden mallissa operaatioista ei tarvitse muodostaa yhtä kokonaisjärjestystä vaan riittää kun kausaalisesti riippuvat operaatiot esitetään todellisessa järjestyksessään.

SPA-sovellukset toteuttavat kausaalisen eheyden luonnostaan, sillä yhteinen tietokanta muodostaa operaatioista yhden järjestyksen, jonka mukaan selainsovellukset voivat havaita operaatiot. Kausaalinen eheys toteuttuu edellisessä esimerkissä, jossa Alice avaa sovelluksen klo 14.00, vaikka peräkkäiseheys ei toteudukaan. Kausaalinen eheys on kuitenkin mahdollista rikkoa mikäli selainsovellus lataa itselleen vain osan päivityksistä. Seuraavassa esimerkissä 7.2 käsitellään huonevarausjärjestelmässä tapahtuvaa kausaalisen eheyden rikkoutumista. Esimerkissä operaatio P2 on kausaalisesti riippuva operaatiosta P1. Käyttäjä 1 päivittää osaa paikallisesta tilastaan ja saa vain operaation P2 tekemän päivityksen, sillä operaatiot P1 ja P2 kohdistuivat eri tauluihin tietokannassa. Kausaalinen eheys ei siis toteudu.

```
L1: Käyttäjä 1 avaa sovelluksen ja lataa sovelluksen
    sen hetkisen tilan kahdella pyynnöllä:
      GET /rooms
      GET /reservations?date=2020-02-02
    Hän näkee vapaana vain huoneen 101.
P1: Käyttäjä 3 poistaa huoneen 105 käytöstä:
      PATCH /rooms?room_id=105
      { disabled: true }
N1: Käyttäjä 2 saa notifikaation, sillä
    hänelle oli varaus huoneeseen 105.
P2: Käyttäjä 2 poistaa varauksensa.
    Tämä operaatio on kausaalisesti riippuva
    operaatiosta P1
L2: Käyttäjä päivittää vielä huonevarausdatan
    pyynnöllä:
      GET /reservations?date=2020-02-02.
    Nyt hän näkee huoneet 101 ja 105 vapaana.
P3: Käyttäjä 1 yrittää varata huoneen 105,
    mutta operaatio epäonnistuu
```



Kuva 7.2: Kausaalisen eheyden rikkoutuminen REST-rajapinnan kanssa

Kausaalinen eheys voi rikkoutua myös selainsovellusten kommunikoidessa suoraan keskenään esimerkiksi WebRTC-yhteyden kautta. Esimerkki 7.3 havainnollistaa tätä. Alice, Billy ja Chloe keskustelevat viestisovelluksella, joka lähettää viestit peer-to-peer -periaattella suoraan laitteesta toiseen.

Chloen kommetti vaikuttaa oudolta, mutta se selittyy tapahtuneella kausaalisen eheyden rikkoneella anomalialla. Alicen ja Chloen laitteiden välinen yhteys katkesi heti Alicen ensimmäisen viestin jälkeen. Bob kuitekin sai Alicen molemmat viestit ja kommentoi helpottuneena. Chloelle kuitenkin näytti siltä, että Bob ilahtui avainten katoamisesta, sillä hän ei nähnyt Alicen jälkimmäistä viestiä.

```
Alicen näkymä:

<Alice> Avaimeni ovat kadonneet!
<Alice> Väärä hälytys, ne löytyikin jo
<Billy> Hienoa!
<Chloe> Billy olet törkeä!
```

```
Chloen näkymä:

<Alice> Avaimeni ovat kadonneet!
<Billy> Hienoa!
<Chloe> Billy olet törkeä!
```

Kuva 7.3: Kausaalisen eheyden rikkoutuminen WebRTC-yhteyden kanssa

7.3.5 Jonoeheys

Jonoeheydeksi [42] (engl. FIFO consistency) kutsutussa mallissa kunkin prosessin kirjoitusoperaatiot on näytettävä oikeassa järjestyksessä. Malli ei ota kantaa eri prosesseissa tapahtuneiden operaatioiden väliseen järjestykseen. SPA-sovelluksissa jonoeheys toteutuu luonnostaan mikäli selainsovellus lähettää omat päivityksensä palvelimelle yksi kerrallaan tai yhteen pyyntöön paketoituna. Huolimattomalla ohjelmoinnilla SPA-sovellus voi kuitenkin rikkoa myös jonoeheyttä. Jos selainsovellus lähettää useita päivityksiä palvelimelle peräkkäin, mutta vastauksia odottamatta, ei päivitysten järjestys ole taattu. HTTP pipelining tekniikalla useita XHR-pyyntöjä voidaan suorittaa FIFO-jonon kautta, mutta se ei ole oletuksena käytössä selaimissa [13].

Peräkkäiseheyden ja kausaalisen eheyden tapaan jonoeheys on helppo rikkoa SPA-sovelluksissa lukemalla palvelimelta dataa paloittain ja eri aikoihin. Tällöin yhdenkin prosessin tekemät operaatiot voidaan lukea väärässä järjestyksessä.

7.4 Kirjoitusoperaatioden konfliktit

Kun samanaikaiset transaktiot asynkronisesti replikoidun tietokannan eri pisteissä päivittävät samaa dataa, on kirjoitusoperaatioiden konfliktin mahdollisuus. Järjestelmän on kyettävä päättämään kumpi ristiriitaisista päivityksistä jää voimaan tai millä muulla keinolla konflikti voidaan ratkaista. Tätä prosessia kutsutaan nimellä konfliktien ratkaiseminen (engl. conflict resolution) ja siihen on olemassa lukuisia menetelmiä. Menetelmistä voidaan erottaa ainakin neljä kategoriaa: konfliktien välttäminen, voittavan arvon valinta, manuaalinen yhdistäminen ja automaattinen yhdistäminen [38].

Yksinkertainen tapa ratkaista konfliktit on estää niiden syntyminen. Jos tietueen muokkaaminen on mahdollista vain yhdessä tietokantapisteessä, konflikteja ei pääse syntymään. Tämä voidaan toteuttaa esimerkiksi Tanenbaumin [57] esittelemällä local-write primary-

backup -menetelmän tapaan. Siinä dataa päivittävä tietokantapiste pyytää tietyn datan kirjoitusoikeuden itselleen ennen päivitysoperaatiota. Operaation jälkeen muutokset propagoidaan muille pisteille ja datan kirjoitusoikeudet palautetaan normaaliksi.

Toinen yksinkertainen tapa konfliktien ratkaisuun on määritellä jonkinlainen keino voimaan jäävän arvon valintaan. Kleppmann [38] listaa seuraavia esimerkkejä keinoista: korkein aikaleima voittaa (engl. last-write-wins), satunnainen valinta, aakkosjärjestys, prosessien prioriteettijärjestys. Näissä kuitenkin on ongelmana datan häviäminen, sillä toinen arvoista ei jää talteen.

Kehittyneemmissä ratkaisuissa dataa ei häviä vaikka konflikteja pääsisi syntymään. CouchDB [8] käyttää mallia, jossa konfliktitilanteessa molemmat arvot tallennetaan ja seuraavan kerran dataa luettaessa molemmat arvot palautetaan sovellukselle. Sovelluksesta voidaan pyytää käyttäjää valitsemaan voimaan jäävä arvo tai se voidaan ratkaista automaattisesti sovelluslogiikan avulla.

Konflikti voidaan ratkaistaan myös heti sen tapahtuessa automaattisilla tekniikoilla. Kleppmann [38] mainitsee seuraavat menetelmät: operational transform, mergeable persistent data structures ja conflict-free replicated datatypes (CRDT). Näiden menetelmien kohdalla tutkimus on osin melko tuoretta ja käytännön implementaatiot eivät ole vielä laajasti vakiinnuttaneet paikkaansa osaksi yleisimpiä tietokantoja.

Shapiro ym. [56] määrittelivät CRDT-tietorakenteiden konseptin vuonna 2011. Ne ovat tietorakenteita, jotka ovat suunniteltu replikoitaviksi useille laitteille ja palvelimille. Niissä on olennaista kyky deterministisesti yhdistää kaksi versiota datasta yhdeksi, jolloin manuaalista konfliktin ratkaisemista ei tarvitse tehdä. Tämän saavuttamiseksi tietorakenteissa käytetään muun muassa versiointia ja vain tiettyjä sallittuja operaatioita. Esimerkiksi kokonaislukulaskuri voidaan toteuttaa niin, että laskurin asettamista tiettyyn arvoon ei sallita. Sen sijaan sallitaan laskurin kasvattaminen tai laskeminen. Kaikki tehdyt operaatiot tallennetaan tietorakenteeseen ja tehtyjen operaatioiden perusteella voidaan määrittää laskurin nykyinen arvo.

SPA-sovelluksien kannalta mielenkiintoisia tekniikoita ovat ainakin kirjoitusoikeuksien jakaminen sekä CRDT:t. Kirjoitusoikeuksien jakamisen implementaatiossa on kuitenkin syytä olla huolellinen, sillä lukitusmenetelmiä käytettäessä on mahdollisuus lukkiumatilanteisiin (engl. deadlock). Kenties lukitukset olisi syytä tehdä jonkin kirjaston avulla eikä osana sovelluslogiikkaa. CRDT-tietorakenteista on olemassa lupaava JavaScript-implementaatio nimeltä Automerge [37]. Tämä mahdollistaa näiden tietorakenteiden hyödyntämisen SPA-sovelluksissa. Tutkielman kirjoittamisen aikana AutoMergen kehitys vai-

kuttaa aktiiviselta, joten se vaikuttaa tekniikalta, jota kannattaa pitää silmällä.

7.5 Havaintoja

ACID-ominaisuuksien noudattaminen SPA-sovelluksissa vaatii huolellisuutta sovelluskehittäjältä. Tämä ilmenee varsinkin operaatioiden atomisuuden varmistamisessa, sillä REST-malli voi ohjata jakamaan operaation useisiin HTTP-pyyntöihin vaikka atomisuuden takaamiseksi ne tulisi suorittaa yhden pyynnön avulla. Pysyvyysvaatimuksen toteuttamiseen SPA-sovelluksilla on hyvät edellytykset web-selainten tallennusrajapintojen avulla. Datan oikeellisuuden turvaamiseen SPA-sovellusten paikallisen tilan osalta jää sen sijaan pitkälti sovelluskehittäjän vastuulle. Tietokantojen käyttämien eheysrajoitteiden kaltaisista mekanismeista ei ainakaan Redux-kirjaston yhteydessä löydy juuri mainintoja. Eristyvyys toteutuu Redux-kirjaston yhteydessä luonnostaan, sillä paikalliseen tilaan tehtävät operaatiot toteutetaan peräkkäisjärjestyksessä.

Jotkin eheysanomaliat tai niiden kaltaiset ilmiöt ovat kuitenkin mahdollisia SPA-sovelluksissakin. Optimistisia päivityksiä käytettäessä on mahdollista sitoutumattoman datan lukeminen. Huolimattomalla ohjelmoinnilla päivitysten ylikirjoittaminen on mahdollista SPA-sovelluksissa, kuten tietokantasovelluksissa yleensä. SPA-sovelluksissa voidaan kuitenkin hyödyntää HTTP-protokollan ETag-tunnisteita tämän anomalian välttämisessä. Toistokelvoton lukeminen ja haamuilmiö eivät ole relavantteja SPA-sovelluksien kannalta, mutta niiden kaltainen tilanne on mahdollinen. Jos SPA-sovellus muodostaa paikallisen tilansa useiden pyyntöjen avulla saadusta datasta, on mahdollista, että paikallinen tila ei vastaa mitään tietokannan eheää tilannevedosta.

Vertailtaessa SPA-sovelluksia eheysmalleihin havaittiin, että keskitetyn tietokannan ansiosta SPA-sovellusten kirjoitusoperaatioista voidaan muodostaa helposti yksi kokonaisjärjestys. Tämä mahdollistaa peräkkäis-, kausaalisen ja jonoeheyden noudattamisen. Ongelmaksi muodostuu vaatimus operaatioiden havaitsemista tietyssä järjestyksessä. Mikäli selainsovellus lataa dataa palvelimelta valikoiden, voidaan rikkoa jokaista mainituista malleista. SPA-sovelluksissa tulisi käyttää jonkinlaista mekanismia operaatioiden havaitsemisjärjestyksen kontrolloimiseksi, jotta hajautettujen tietokantojen eheysmalleja voitaisiin hyödyntää. Tällaisen mekanismin avulla peräkkäiseheys olisi luonteva malli SPA-sovelluksille.

Lopuksi käsiteltiin vielä kirjoitusoperaatioiden konfliktien ratkaisemista. SPA-sovelluksissa samaa dataa voidaan päivittää useassa paikassa yhtäaikaa ja jos ha-

lutaan sallia sovelluksen toiminta myös offline-tilassa on tärkeää kyetä ratkaisemaan mahdollisesti syntyvät ristiriitaisuudet. Naiivit menetelmät johtavat datan menettämiseen esimerkiksi ylikirjoittumisen takia. CouchDB-tietokannan käyttämä menetelmä, jossa konfliktin syntyessä molemmat arvot tallennetaan, voi olla käyttökelpoinen monille SPA-sovelluksille. Hienostunein menetelmä on kuitenkin konfliktien automaattinen ratkaiseminen esimerkiksi CRDT-tietorakenteiden avulla. Niiden osalta tutkimus on vielä melko tuoretta ja SPA-sovellusten kannalta erityisen mielenkiintoisia ovat niistä kehiteltävät JavaScript-implementaatiot.

8 Pohdinta ja johtopäätökset

Tässä luvussa vastataan johdannossa esitettyihin tutkimuskysymyksiin, arvioidaan tutkimuksen luotettavuutta sekä pohditaan mahdollisia jatkotutkimusaiheita. Tiivistettynä voidaan sanoa, että tilanhallinta SPA-sovelluksissa muistuttaa replikoitua multi-master tietokantaa. Tätä rinnastusta käyttäen löydettiin replikoiduista tietokannoista useita ideoita SPA-sovellusten selaimen ja palvelimen välisen kommunikaation sekä eheyden hallinnan kehittämiseen. Tutkielmassa esitetty analyysi on melko epäformaalia, mutta tutkimalla olemassa olevien sovellusten implementaatioita olisi mahdollista validoida tutkimuksen tuloksia. Lisäksi tutkielmassa tehdyt havainnot toivat esille muutamia muita jatkotutkimusaiheita.

8.1 Tulosten analysointi

TK1: Millaisia hajauttujen tietokantojen piirteitä SPA-sovelluksilla on?

Ensimmäisen tutkimuskysymyksen tutkimista varten esiteltiin erilaisia hajauttuja tietokantoja ja niiden tavoittelemia ominaisuuksia. Vertailemalla SPA-sovelluksia näihin ominaisuuksiin havaittiin, että SPA-mallin avulla tavoitellaan samoja ominaisuuksia kuin replikoiduilla tietokannoilla ja myöskin datan hajauttamisen malli on lähimpänä replikoituja tietokantoja. Molemmat tavoittelevat lukuoperaatioiden vasteajan pienentämistä ja tämä saavutetaan ylläpitämällä datasta useita kopioita. Kopiot sijoitetaan usein maantieteellisesti hajauttaen niin, että data on riittävän lähellä asiakasta hyväksyttävän vasteajan saavuttamiseksi. Lisäksi SPA-sovelluksissa sallitaan kirjoitusoperaatiot useisiin pisteisiin, joten asetelma muistuttaa replikoitua multi-master tietokantaa.

TK2: Käytetäänkö SPA-sovelluksissa samankaltaisia datan replikointiin liittyviä menetelmiä kuin replikoiduissa tietokannoissa ja voisiko tietokantojen replikointimenetelmistä ottaa mallia SPA-sovelluksissa?

Tätä tutkimuskysymystä tutkittiin poimimalla replikoitujen tietokantojen käyttämistä replikointimenetelmistä erilaisia kuvailevia ominaisuuksia malliksi, jolla menetelmiä voidaan luokitella. Tämän mallin avulla vertailtiin tietokantojen replikointiprotokollia erilaisiin web-sovellusten tiedonsiirtoon liittyviin tekniikoihin.

Vertailun tuloksena havaittiin, että web-sovellusten tyypillisesti käyttämät tekniikat ovat melko erilaisia kuin replikoiduissa tietokannoissa. Tietokannoissa replikointi tehdään usein automatisoidusti ja päivitykset levitetään push-mallilla. Tyypillinen HTTP-pyyntöjä käyttävä SPA-sovellus toimii päinvastoin. Selain saa päivitykset pull-mallilla ja sovelluskehittäjän on implementoitava replikointilogiikka itse. WebSocket-protokolla kuitenkin mahdollistaa päivitysten työntämisen selaimeen, joten tässä kohtaa web-sovelluksissa voisi ottaa mallia tietokannoista ja siirtyä enemmän push-malliin. PouchDB-tietokanta taas osoittaa, että automaattinen replikointi on mahdollista myös selaimen ja palvelimen välillä. PouchDB:n suosiota kuitenkin rajannee sen sidonnaisuus CouchDB-tietokantaan. Vastaava mahdollisuus automaattiseen replikointiin yhdistettynä suositumpaan tietokantaan voisi olla erittäin mielenkiintoinen ratkaisu SPA-sovelluksiin.

TK3: Miten tietokantojen kontekstissa määritellyt erilaiset eheyshaasteet ja eheysmallit ilmenevät SPA-sovelluksissa?

Viimeisen tutkimuskysymyksen tarkastelu voidaan jakaa neljään osa-alueeseen. Perinteisesti tietokannat turvaavat datan eheyden transaktioiden avulla, joten SPA-sovellusten eheyden hallinnan arviointiin käytettiin ensin transaktioiden ACID-ominaisuuksia ja sitten tunnettuja eristyvyysanomalioita. Kolmanneksi tutkittiin aihetta hajauttujen tietokantojen eheysmallien kautta ja lopuksi kirjoitusoperaatioiden konfliktien ratkaisemisen näkökulmasta.

ACID-ominaisuuksia analysoitaessa tehtiin muutamia hyödyllisiä havaintoja. SPA-sovelluksille on olemassa menetelmiä atomisuuden ja pysyvyyden toteuttamiseen. Atomisuuden varmistaminen vaatii kuitenkin sovelluskehittäjältä tarkkaavaisuutta. Datan oikeellisuuden turvaamiseen esimerkiksi eheysrajoitteiden avulla ei sen sijaan vaikuttaisi juuri olevan yleisiä käytänteitä tai kirjastoja, joten SPA-sovelluksissa siitä huolehtiminen jää pitkälti sovelluskehittäjän vastuulle.

Myös eristyvyysanomalioiden tarkastelu osoittautui hyödylliseksi. Niiden tai niiden kaltaisten ilmiöiden havaittiin olevan mahdollisia myös SPA-sovelluksissa palvelinpyyntöjen yhteydessä. Huolimattomalla ohjelmoinnilla esimerkiksi päivitysten ylikirjoittaminen on mahdollista. Tämän anomalian välttämiseen on kuitenkin esimerkiksi HTTP-protokollassa sisäänrakennettu menetelmä. Toistokelvottoman lukemisen anomalian tarkastelu auttoi havaitsemaan vastaavan ilmiön SPA-sovelluksissa. Kun SPA-sovellus muodostaa paikallisen tilansa useiden pyyntöjen avulla saadusta datasta, on paikallinen tila sekoitus eri ajanhetkillä tietokannassa olleesta datasta.

Hajautettujen tietokantojen eheysmalleista peräkkäiseheys, kausaalinen eheys ja jonoe-

heys ovat tutkielman tarkastelun perusteella selkeimmin soveltuvia SPA-sovelluksille. Kun eri käyttäjät kommukoivat yhden keskitetyn tietokannan kautta, on helppo muodostaa jaettu käsitys operaatioiden tapahtumisjärjestyksestä. Ongelmaksi kuitenkin muodostuu vaatimus operaatioiden havaitsemisesta tietyssä järjestyksessä. SPA-sovellukset voivat ladata dataa palvelimelta valikoiden ja tällöin mainittujen mallien mukainen eheys ei toteudu. Eheysmalleja koskevan analysoinnin keskeisenä tuloksena havaittiin siis, että SPA-sovelluksissa tulisi käyttää jonkinlaista mekanismia operaatioiden havaitsemisjärjestyksen kontrolloimiseksi, jotta hajautettujen tietokantojen eheysmalleja voitaisiin hyödyntää.

Lisäksi havaittiin, että kirjoitusoperaatioiden konfliktien ratkaisemiseen käytettävät CRDT-tietorakenteet ovat lupaava menetelmä myös SPA-sovelluksille, sille niiden tutkimus on aktiivista ja niistä on kehitetty JavaScript-implementaatio, joka mahdollistaa niiden käyttämisen web-selaimessa.

8.2 Tulosten validiteettitarkastelu

Tämän tutkielman tutkimusotetta voidaan kuvailla design science ja tapaustutkimus -tyyppiseksi. Esitetyt vertailut ja analyysit ovat laadullista tutkimusta. Näihin tutkimustapohin liittyvät ongelmat ovat läsnä myös tässä tutkielmassa.

Laadullista tutkimusta värittävät tutkijan kokemukset ja subjektiivinen tunne. Vahvistusharha saa tutkijan suosimaan hypoteesiaan tukevia havaintoja ja tuloksia. Tässä tutkielmassa vahvistusharha on voinut vaikuttaa sekä tiedon keräämiseen että analysointiin. Aineiston keräämisen puutteellisuus ja edustavuuden puute liitetään myös tapaustutkimus-strategiaan [1]. Lähteiden valinnassa ja käsiteltävien asioiden rajaamisessa toinen henkilö tutkimuksen tekijänä olisi luultavasti päätynyt erilaisiin valintoihin. Lisäksi on vaikea sanoa varmuudella esimerkiksi tiettyjen tekniikoiden, kuten eheysrajoitteiden, esiintymättömyydestä SPA-sovellusten kontekstissa. Voi olla, että jokin tekniikka onkin käytössä, mutta esimerkisi käytetyt hakutermit ovat olleet vääriä ja oikeat kirjastot jäävät löytymättä. Luotettavampi katsaus käytettyihin tekniikoihin vaatisi perehtymisen riittävän laajaan otokseen todellisia ohjelmistoja ja niiden tekniikoiden kartoittamiseen. Tämän tutkielman yleistettävyyteen vaikuttavat myös SPA-sovelluksen rakenteesta ja yksinkertaisuudesta sekä käytetyistä teknologioista tehdyt oletukset.

Vertailu ja ominaisuuksien toteutumisen arviointi melko vapaamuotoista eikä esimerkiksi mittauksiin kvantitaviista tutkimusta tai formaalimpaa todistuksiin perustuvaa päättelyä. Tutkimuksen toistaminen vaatisi kenties toisen samoihin tutkimuskysymyksiin vastaavan

tutkielman kirjoittamista toisen henkilön tai useiden muiden henkilöiden toimesta. Tulosten vahvistaminen voisi siis olla näin mahdollista toistamalla tutkimus, mutta se olisi melko työlästä.

Tutkielma kuitenkin vaikuttaisi saavuttavan design science -menetelmää käyttäville tutkimuksille tyypillisen [59] tavoitteen: tuottaa uutta hyödyllistä tietoa.

8.3 Johtopäätöksiä ja aiheita jatkotutkimukselle

Selkeä eroavaisuus SPA-sovellusten ja replikoitujen tietokantojen välillä oli replikoinnin automaattisuudessa. PouchDB on näyttänyt, että automaattisen replikoinnin toteuttaminen on mahdollista myös selaimen ja palvelimen välillä. Tietyt PouchDB:n ominaisuudet ovat kenties hidastaneet sen suosion kasvua, joten olisi mielenkiintoista lähteä toteuttamaan sama idea esimerkiksi PostgreSQL-tietokantaa käyttäen. PouchDB:n replikointi hyödyntää suoraan CouchDB:n replikointirajapintaa. Samoin voisi kenties tehdä PostgreSQL:n kohdalla ja hyödyntää sen loogisen replikoinnin rajapintaa.

Loogisten lokien käytöstä voisi olla hyötyä myös selaimessa tehtävään persistointiin. Kokonaisen tilaobjektin persistoinnin sijaan selaimessa voitaisiin persistoida relaatiotietokantojen käyttämien transaktiolokien tapaan tiiviitä lokeja tapahtuneista operaatiosta. Näistä lokeista voitaisiin, kenties tilannevedoksiin yhdistettynä, palauttaa sovelluksen tila esimerkiksi vahingossa tapahtuneen selainikkunan sulkemisen jälkeen. Tällaiset lokit ovat nopeita kirjoittaa levylle verrattuna suureen JSON-muotoisen tilaobjektiin, joka joudutaan serialisoimaan merkkijonoksi ennen tallentamista. Parhaimmillaan samaa lokiformaattia olisi mahdollista käyttää selaimessa tehtävässä persistoinnissa, replikoinnissa sekä palvelimen tietokantaoperaatioissa.

ACID-ominaisuuksia tarkastellessa syntyi havainto, että selaimessa olevan datan ACID-aknonyymin mukaiseen oikeellisuuteen liittyen löytyy todella vähän materiaalia tai kirjastoja. Paikallisen tietokannan oikeellisuus esimerkiksi uniikkiusrajoitteiden avulla on kenties jäänyt vähemmälle huomiolle, kun palvelimen tietokanta hoitaa sen lopulta. Tietokantojen käyttämien eheysrajoitteiden kaltaisista tekniikoista voi olla hyötyä varsinkin, jos lähdetään muuttamaan palvelimen ja selaimen välistä painopistettä progressive web app tai local-first -paradigmojen suuntaan ja sallitaan enemmän offline-tilassa tehtäviä päivityksiä. ACID-ominaisuuksien osalta olisi mielenkiintoista tutkia kuinka laajassa käytössä JSON PATCH -notaatio on vai tehdäänkö useita resursseja muokkaavat operaatiot usein erillisillä HTTP-pyynnöillä, jolloin niiden atomisuudesta ei ole takeita.

Myös ETag-tunnisteiden tai muiden lost-update -anomalian estävien mekanismien käytön yleisyyttä olisi myös mielenkiintoista selvittää.

HTTP-protokolla ja REST-rajapinnat ovat muodostuneet oletusarvoiseksi tavaksi siirtää dataa selaimen ja sovelluspalvelimen välillä. Niissä data kuitenkin liikkuu pull-mallilla palvelimelta selaimeen - päinvastoin kuin replikoiduissa tietokannoissa missä data liikkuu tyypillisesti push-mallia käyttäen. WebSocket-protokolla mahdollistaa datan työntämisen palvelimelta selaimeen, mutta WebSocket-protokollan pariksi ei vaikuta muodostuneen samanlaista REST-rajapintojen kaltaista oletusarvoista mallia. Mielenkiintoinen tutkimusaihe olisi selvittää onko WebSocket-protokollaa käyttävillä sovelluksilla käytössä jotain REST-arkkitehtuurin kaltaisia malleja vai käytetäänkö niissä sovellusspesifejä ratkaisuja. Yleisten mallien puuttuessa sellaisen kehittäminen voisi olla erittäin hyödyllistä.

Uusien selaimessa käytettävien tekniikoiden ja mallien kehittämiseen kannustaa tutkielman alussa tehty huomio web-tekniikoiden kehittymisestä. Monet tekniikat, kuten Ajax ja SPA, ovat nousseet pinnalle vasta vuosia sen jälkeen, kun niiden mahdollistavat tekniikat olivat käytettävissä selaimissa. Tämän perusteella voisi ennustaa, että suhteellisiin uusiin standardeihin, kuten IndexedDB ja WebSocket, liittyen tullaan näkemään uusia menetelmiä ja suureen suosioon kasvavia kirjastoja.

9 Yhteenveto

Tutkielmassa perehdyttiin SPA-sovelluksiin, joista on tullut todella suosittu tapa rakentaa selainsovelluksia. SPA-mallissa selaimessa säilytetään sovelluksen tilaa tietokantamaisessa muodossa. Varsinkin yksinkertaisissa sovelluksissa sovelluspalvelimen tehtäväksi jää lähinnä datan siirtäminen tietokannan ja selaimen välillä. Asetelma muistuttaa hajautettua tietokantaa.

Tämän rinnastuksen pohjalta tutkielmassa vertailtiin SPA-sovelluksia hajautettuihin tietokantoihin. Vertailu keskittyi erityisesti datan replikointiin kahden pisteen välillä sekä datan eheyden hallintaan. Näitä aiheita on tutkittu hajauttujen tietokantojen kontekstissa vuosikymmeniä ja tämän tutkielman tarkoituksena oli selvittää, mitä vuosien tutkimuksen ja kehityksen tuloksena syntyineistä periaatteista ja tekniikoista kannattaisi soveltaa myös SPA-sovelluksissa.

Vertailemalla SPA-sovelluksia erilaisiin hajautettuihin tietokantoihin ja niiden tavoittelemiin ominaisuuksiin havaittiin, että SPA-sovelluksilla on samoja piirteitä kuin replikoiduilla tietokannoilla, jotka sallivat kirjoitusoperaatioiden suorittamisen useissa eri pisteissä. Molemmissa on tavoitteena pienentää vasteaikaa dataa luettaessa ja tämä saavutetaan replikoimalla dataa useisiin pisteisiin niin, että data on lähempänä käyttäjää.

Seuraavaksi SPA-sovellusten käyttämiä selaimen ja palvelimen väliseen tiedonsiirtoon liittyviä tekniikoita vertailtiin replikoitujen tietokantojen käyttämiin replikointiprotokolliin. Vertailun tuloksena havaittiin, että web-sovellusten tyypillisesti käyttämät tekniikat ovat melko erilaisia kuin replikoiduissa tietokannoissa. Tietokannoissa replikointi tehdään usein automatisoidusti ja päivitykset levitetään push-mallilla. Tyypillinen HTTP-pyyntöjä käyttävä SPA-sovellus toimii päinvastoin. Selain saa päivitykset pull-mallia käyttäen ja sovelluskehittäjän on implementoitava replikointilogiikka itse.

Lopuksi tutkittiin miten tietokantojen kontekstissa määritellyt erilaiset eheyteen liittyvät haasteet ja eheyden hallintamenetelmät ilmenevät SPA-sovelluksissa. Vertailun tuloksena havaittiin, että SPA-sovelluksissa on hyvät edellytykset eheyteen liittyvien ongelmien välttämiseen, mutta huolimattomalla ohjelmoinnilla voidaan törmätä useisiin erilaisiin anomalioihin. Esimerkiksi muiden käyttäjien tekemien päivitysten ylikirjoittaminen on mahdollista melko helposti, mutta tämän välttämiseksi on olemassa keinoja. Samoin useiden resurssien päivittäminen atomisesti vaatii huolellisuutta. Mikäli operaatioiden ato-

misuudesta ei huolehdita, saattaa tietokantaan päätyä keskeneräisiä operaatioita. SPA-sovelluksissa voidaan kohdata epäeheää dataa myös selainsovelluksen käyttämän valikoivan datan noutamisen takia. Kun selainsovellus rakentaa paikallisen tilansa lataamalla dataa palvelimelta useassa osassa ja eri ajanhetkinä, ei selaimen paikallinen tila vastaa välttämättä mitään eheää tilannekuvaa palvelimella olevasta datasta. Tällöin paikallisessa datassa voi olla ristiriitaisuuksia.

Tutkielman havaintojen perusteella heräsi useita ideoita mahdollisiksi jatkotutkimusaiheiksi. Automaattisen replikoinnin tarjoavan tilanhallintamekanismin kehittäminen SPA-sovelluksille olisi potentiaalisesti todella hyödyllinen projekti. Lisäksi selaimessa tehtävään sovelluksen tilan paikalliseen tallentamiseen löydettiin kehittämismahdollisuuksia. Myös paikallisen tilan eheyden varmistamismekanismien kehittämisessä olisi tehtävää, sillä sellaisia ei juuri SPA-sovelluksille ole. Erittäin mielenkiintoinen tutkimusaihe olisi myös perehtyä joukkoon todellisia SPA-sovelluksia ja tutkia esiintyykö niissä eheysanomalioita ja onko esimerkiksi päivitysten ylikirjoittamisen estäviä menetelmiä käytössä vai luotetaanko eheyden hallinnassa enemmänkin hyvään tuuriin.

Kirjallisuus

[1] E. Aarnos. *Ikkunoita tutkimusmetodeihin. 1, Metodin valinta ja aineistonkeruu : virikkeitä aloittelevalle tutkijalle*. 5., uudistettu painos. Jyväskylä: PS-kustannus, 2018.

[2] D. Abramov. *Documentation*. 2020. URL: https://redux.js.org (viitattu 11. 04. 2020).

[3] D. Abramov. *Ecosystem*. 2020. URL: https://redux.js.org/recipes/structuring-reducers/normalizing-state-shape (viitattu 07. 01. 2020).

[4] D. Abramov. *Ecosystem*. 2020. URL: https://redux.js.org/recipes/usage-with-typescript (viitattu 07. 01. 2020).

[5] D. Abramov. *Ecosystem*. 2020. URL: https://redux.js.org/introduction/ecosystem/ (viitattu 07. 01. 2020).

[6] D. Abramov. *Three Principles*. 2020. URL: https://redux.js.org/introduction/three-principles (viitattu 06. 04. 2020).

[7] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli ja P. W. Hutto. "Causal memory: Definitions, implementation, and programming". *Distributed Computing* 9.1 (1995), s. 37–49.

[8] *Apache CouchDB 3.0.0 Documentation*. 2020. URL: http://docs.couchdb.org/en/stable/intro/overview.html (viitattu 11. 04. 2020).

[9] ApolloGraphQL. *Optimistic UI*. 2020. URL: https://www.apollographql.com/docs/react/performance/optimistic-ui/ (viitattu 02. 02. 2020).

[10] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil ja P. O'Neil. "A Critique of ANSI SQL Isolation Levels". *SIGMOD Rec.* 24.2 (toukokuu 1995), s. 1–10. ISSN: 0163-5808. DOI: 10.1145/568271.223785. URL: https://doi.org/10.1145/568271.223785.

[11] E. A. Brewer. "Towards Robust Distributed Systems (Abstract)". Teoksessa: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. Association for Computing Machinery, 2000, s. 7. ISBN: 1581131836. DOI: 10.1145/343477.343502.

[12] C. G. Bruce G. Lindsay Patricia Griffiths Selinger. "Notes on distributed databases" (1979).

[13] *Connection management in HTTP/1.x*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Connection_management_in_HTTP_1.x (viitattu 11. 04. 2020).

[14] C. J. Date. *An introduction to database systems*. 6th. Addison-Wesley publishing company, 1994. ISBN: 0201824582.

[15] Facebook. *Glossary of React Terms*. 2019. URL: https://reactjs.org/docs/glossary.html (viitattu 27. 12. 2019).

[16] I. Fette ja A. Melnikov. *The WebSocket Protocol*. RFC 6455. RFC Editor, 2011. URL: https://tools.ietf.org/html/rfc6455.

[17] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. 2000.

[18] D. Flanagan. *JavaScript: The Definitive Guide Activate Your Web Pages*. 6th. O'Reilly Media, Inc., 2011. ISBN: 9780596805524.

[19] J. J. Garret. *Ajax: A New Approach to Web Applications*. 2005. URL: https://pdfs.semanticscholar.org/c440/ae765ff19ddd3deda24a92ac39cef9570f1e.pdf (viitattu 27. 12. 2019).

[20] S. Ghosh. *Distributed Systems: An Algorithmic Approach, Second Edition*. 2nd. Chapman & Hall/CRC, 2014. ISBN: 1466552972.

[21] *GraphQL*. URL: https://graphql.org/ (viitattu 14. 04. 2020).

[22] J. Gray et al. "The transaction concept: Virtues and limitations". Teoksessa: *VLDB*. Vol. 81. 1981, s. 144–154.

[23] J. Gray, P. Helland, P. O'Neil ja D. Shasha. "The Dangers of Replication and a Solution". Teoksessa: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD '96. Montreal, Quebec, Canada: Association for Computing Machinery, 1996, s. 173–182. ISBN: 0897917944. DOI: 10.1145/233269.233330. URL: https://doi.org/10.1145/233269.233330.

[24] T. Haerder ja A. Reuter. "Principles of Transaction-Oriented Database Recovery". *ACM Comput. Surv.* 15.4 (joulukuu 1983), s. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291. URL: https://doi.org/10.1145/289.291.

[25] M. P. Herlihy ja J. M. Wing. ”Linearizability: A Correctness Condition for Concurrent Objects”. *ACM Trans. Program. Lang. Syst.* 12.3 (heinäkuu 1990), s. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972. URL: https://doi.org/10.1145/78969.78972.

[26] I. Hickson. *Web Database W3C Working Draft 29 October 2009*. 2009. URL: https://www.w3.org/TR/2009/WD-webdatabase-20091029/.

[27] I. Hickson. *Web SQL Database W3C Working Group Note 18 November 2010*. 2010. URL: http://www.w3.org/TR/2010/NOTE-webdatabase-20101118/ (viitattu 02. 01. 2020).

[28] I. Hickson. *Web Storage W3C Recommendation 30 July 2013*. 2013. URL: https://www.w3.org/TR/2013/REC-webstorage-20130730/ (viitattu 02. 01. 2020).

[29] I. Hickson. *Web Storage W3C Working Draft 23 April 2009*. 2009. URL: https://www.w3.org/TR/2009/WD-webstorage-20090423/ (viitattu 02. 01. 2020).

[30] P. Hunt. *Why did we build React?* 2013. URL: https://reactjs.org/blog/2013/06/05/why-react.html (viitattu 11. 04. 2020).

[31] *Information technology — Database languages — SQL*. Standard. Geneva, CH: International Organization for Standardization, 1992.

[32] M. A. Jadhav, B. R. Sawant ja A. Deshmukh. ”Single page application using angularjs”. *International Journal of Computer Science and Information Technologies* 6.3 (2015), s. 2876–2879.

[33] C. Jennings, H. Boström ja J.-I. Bruaroey. *WebRTC 1.0: Real-time Communication Between Browsers*. 2019. URL: https://www.w3.org/TR/2019/CR-webrtc-20191213/ (viitattu 03. 05. 2020).

[34] P. R. Johnson ja R. H. Thomas. *The Maintenance of Duplicate Databases*. RFC 677. RFC Editor, 1975. URL: https://tools.ietf.org/html/rfc677.

[35] A. van Kesteren, J. Aubourg, J. Song ja H. R. M. Steen. *XMLHttpRequest Level 1*. 2016. URL: https://www.w3.org/TR/XMLHttpRequest/ (viitattu 11. 04. 2020).

[36] M. Kleppmann. ”A Critique of the CAP Theorem”. *CoRR* abs/1509.05393 (2015). arXiv: 1509.05393. URL: http://arxiv.org/abs/1509.05393.

[37] M. Kleppmann. *Automerge*. 2020. URL: https://github.com/automerge/automerge (viitattu 30. 03. 2020).

[38] M. Kleppmann. *Designing Data-Intensive Applications*. O'Reilly, 2017. ISBN: 978-1-4493-7332-0.

[39] M. Kleppmann, A. Wiggins, P. van Hardenberg ja M. McGranaghan. "Local-First Software: You Own Your Data, in Spite of the Cloud". Teoksessa: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2019. Athens, Greece: Association for Computing Machinery, 2019, s. 154–178. ISBN: 9781450369954. DOI: 10.1145/3359591.3359737. URL: https://doi.org/10.1145/3359591.3359737.

[40] H. F. Korth ja A. Silberschatz. *Database System Concepts*. 2nd. USA: McGraw-Hill, Inc., 1990. ISBN: 0070447543.

[41] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". *Commun. ACM* 21.7 (heinäkuu 1978), s. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

[42] R. J. Lipton ja J. S. Sandberg. "Pram: a scalable shared memory". Teoksessa: 1988.

[43] M. Mahemoff. *Single-page application*. 2008. URL: https://en.wikipedia.org/w/index.php?title=Single-page_application&oldid=232811237 (viitattu 30. 12. 2019).

[44] N. Mehta. *Indexed Database API W3C Working Draft 05 January 2010*. 2010. URL: https://www.w3.org/TR/2010/WD-IndexedDB-20100105/ (viitattu 02. 01. 2020).

[45] N. Mehta, J. Sinking, E. Graff, A. Popescu, J. Orlow ja J. Bell. *Indexed Database API W3C Recommendation 08 January 2015*. 2015. URL: https://www.w3.org/TR/2015/REC-IndexedDB-20150108/ (viitattu 02. 01. 2020).

[46] M. Mikowski ja J. Powell. *Single Page Web Applications: JavaScript End-to-end*. 1st. Greenwich, CT, USA: Manning Publications Co., 2013. ISBN: 9781617290756.

[47] I. Minar. *Release v0.9.0*. 2010. URL: https://github.com/angular/angular.js/releases/tag/v0.9.0 (viitattu 02. 01. 2020).

[48] H. Nielsen ja L. Daniel. *Detecting the Lost Update Problem Using Unreserved Checkout*. Tekninen raportti. W3C, 1999.

[49] J. Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 0125184050.

[50] P. O'Shannessy. *Initial public release*. 2013. URL: https://github.com/facebook/react/releases/tag/v0.3.0 (viitattu 02. 01. 2020).

[51] S. Overflow. *Stack Overflow Developer Survey 2016*. 2016. URL: https://insights.stackoverflow.com/survey/2016#technology (viitattu 30. 12. 2019).

[52] E. P. Bryan ja E. M. Nottingham. *JavaScript Object Notation (JSON) Patch*. RFC 6902. RFC Editor, 2013. URL: https://tools.ietf.org/html/rfc6902.

[53] Postgres. *Comparison of Different Solutions*. 2020. URL: https://www.postgresql.org/docs/12/different-replication-solutions.html (viitattu 17. 01. 2020).

[54] *PouchDB*. URL: https://pouchdb.com/ (viitattu 14. 04. 2020).

[55] A. Russell. 2005. URL: https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/ (viitattu 02. 01. 2020).

[56] M. Shapiro, N. Preguiça, C. Baquero ja M. Zawirski. "Conflict-Free Replicated Data Types". Teoksessa: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, s. 386–400. ISBN: 9783642245497.

[57] A. S. Tanenbaum ja M. Van Steen. *Distributed Systems*. Pearson Education, 2013. ISBN: 1292025522.

[58] *The polyglot developer*. Kesäkuu 2016. URL: https://www.thepolyglotdeveloper.com/2016/06/tpdp-episode-6-pouchdb-usefulness-browser-based-development/.

[59] J. E. Van Aken. "Management Research as a Design Science: Articulating the Research Products of Mode 2 Knowledge Production in Management". *British Journal of Management* 16.1 (2005), s. 19–36. DOI: 10.1111/j.1467-8551.2005.00437.x.

[60] *Web APIs*. URL: https://developer.mozilla.org/en-US/docs/Web/API (viitattu 11. 04. 2020).